Technische Universität München

Institut für Informatik

An Analyzer Tool for Reducing the Costs of Updates in a Heterogeneous Aftersales Database

Ingo Schneider

Technische Universität München

Institut für Informatik

Diplomarbeit

An Analyzer Tool for Reducing the Costs of Updates in a Heterogeneous Aftersales Database

Ingo Schneider

schneidi@in.tum.de

Aufgabensteller:	Prof. Bernd Brügge, Ph.D.
Betreuer:	Günter Teubner
Abgabedatum:	15.11.98

Ich versichere, daß ich diese Diplomarbeit selbstständig verfaßt und
nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Contents

Chapter 1

Introduction

1.1 Motivation

Within the scope of the PAID project, a Platform for Active Information Dissemination will be developed. This project was launched by the GFS[1] consortium, which is a partnership of major IT providers, a research group, and, as the major client, the Daimler-Benz user group. The PAID project is a collaboration between the TU Munich, the Carnegie Mellon University in Pittsburgh, and Mercedes-Benz.

The framework developed by the PAID project will be a proof-of-concept prototype of its system architecture. The information that is handled by this prototype will be the Mercedes-Benz aftersales information, which are for example spare-parts catalogues, service documents, and information about individual vehicles. The aftersales data is currently generated and maintained at the Mercedes-Benz headquarters. It should be distributed to the dealers and workshops all over the world through an adaptive, selective, multicast network developed within that project.

In the worst case, there are only low-bandwidth modem lines connecting the dealers and workshops to the network. The central aftersales database currently covers more than 30 Gigabytes, which is too big as that major parts could be distributed through these low-bandwidth lines. Instead, if changes occur on that database, a minimal amount of data should be distributed. This was the major task of this thesis: to developed an analyzer tool which generates these minimal incremental updates which can then be distributed through the PAID network.

1.2 The Task

As with all software projects, before starting designing and implementing the analyzer tool, the exact requirements had to be found, though some requirements were given: the analyzer tool should be extensible for new types of data; it should be written in the Java programming language; and concepts should be provided for handling exceptional situations.

As a major subtask, the processes of authoring and distributing aftersales information within Mercedes-Benz had to be investigated. Of course, the data structures and coherence had to be examined as well, since they define the input of the analyzer tool. These tasks turned out to be very time consuming as the knowledge about aftersales data and the associated processes is distributed among the staff and departments at Mercedes-Benz and debis[2]. Also, there was no documentation of the

[1]GFS = Global Field Support

[2]debis is a subsidiary company of the Daimler-Benz group delivering IT services.

processes and data structures found that was suitable to work oneself in.

Since the PAID project wasn't yet launched when this thesis was started, it had to be figured out as well which output the analyzer tool should generate. Should it produce some form of updates in a specialized language, should it just feed changed data into a commercial replication mechanism, should it just mark which data had been changed, ... ?

Another question is, why such a tool was actually necessary. Why not use one of those many commercially available database replication techniques for the PAID project ?

The answers to these questions can be found in this thesis, the next section will give an overview of its structure. The main focus of this thesis is the design and implementation of the analyzer tool and a description of the selected aftersales databases. Additionally, a concept for distributing changes, called 'Smart Updates', will be introduced as a proposal for the PAID project. A rudimentary implementation of that concept for the selected parts of the aftersales databases is also presented together with the implementation of the analyzer tool.

1.3 Overview

The following is an overview of the chapters included in this thesis:

Chapter 2, StarNetwork and the PAID Project describes the current distribution channels for aftersales information, and the problems with those which led to the development of StarNetwork. It further explains which improvements are hoped to be brought to StarNetwork by the PAID project. It also tells which particular parts of the aftersales information were selected as reference databases for the project.

Chapter 3, Origination of aftersales data describes the authoring systems for the aftersales information used within the MercedesBenz headquarters. It also explains how the data makes its way into the central StarNetwork database.

Chapter 4, Datamovements within the PAID Network introduces my concept for distribution of changes called 'Smart Updates'. First, some requirements for data replication systems are elaborated. Then the major database replication techniques available are presented, and it is explained why those do not fulfill the requirements. After that, it is shown how the proposed concept meets the requirements. The rudimentary implementation of this concept produced within the scope of this thesis is presented at the end of that chapter.

Chapter 5, The Aftersales Data describes the aftersales data structures of the selected parts of the database in all detail. It also shows how the data is converted to the new format used by the StarNetwork applications. It is explained how the data can be subdivided and customized to the needs of the individual enduser as well.

Chapter 6, Analyzer Design and Implementation describes the architecture of the analyzer tool. Also, the updates for the individual parts of the aftersales database and their generation is described in detail within that chapter.

Chapter 7, Installation and Usage explains how the developed tools are used together with the sample data provided.

Chapter 2

The PAID Project

This chapter describes the current distribution channels for aftersales information, and the problems with those which led to the development of StarNetwork. This is a unified application framework for aftersales applications, which will make the aftersales information accessible via the internet. It will be explained which improvements are hoped to be brought to StarNetwork by the PAID project.

2.1 As-is Situation

Within Mercedes-Benz, aftersales information is created and distributed by different departments. The major information sources today are service, parts, and vehicle documentation. Depending on the type of information the company utilizes a variety of different distribution channels for aftersales information. Table 2.1 shows the main aftersales information types, the respective end user applications and distribution channels.

Information Type	Application	Distribution Channel
Service Information	WIS	CD−ROM, PAPER, Microfiche
Diagnosis Information	DAS	CD−ROM, PAPER, Microfiche
Parts Information	EPC	CD−ROM, PAPER, Microfiche
Vehicle Information	FDOK	ONLINE, CD−ROM
Car Configuration Data	MBKS	CD−ROM
Standard Texts & Flat Rates	ASRA	CD−ROM, FILE
Damage Codes	VEGA	ONLINE, CD−ROM

Table 2.1: Current distribution channels for aftersales information

These distribution channels are typically very reliable but also very slow and inefficient. For instance the distribution of service, parts, and vehicle information to the worldwide Mercedes-Benz sales organization is done via a monthly published set of 15 CD-ROMs. This information is already partially outdated when it gets to the dealer.

Today and in the near future Mercedes-Benz is extending its business in terms of new product lines (A-Class, M-Class, etc.) and new models of already existing product lines (S-Class, etc.). The amount of aftersales information is increasing due to the introduction of these new products. With the introduction of new aftersales information systems additional information distribution channels are created, which finally lead to a proliferation of information distribution channels. All this makes the information management process (creation, publishing, distribution, etc.) from a technology and management point of view more complex and expensive.

Today's information distribution channels are too slow and inflexible to meet these demanding business requirements.

2.2 StarNetwork

Mercedes-Benz is currently developing the StarNetwork application framework. This will be a common framework for the various types of aftersales applications mentioned in the last chapter.

It is designed as a client-server application, using Java as the implementation language for both the client and the server. The client part, which runs within an internet browser, accesses one of the servers remotely via the Mercedes-Benz intranet, extranet, or via the internet.

The databases storing the aftersales information are only accessible to the servers. These are located at the Daimler-Benz headquarters and at the so-called MLCs, which provide the servers for major distribution areas. An example for this is MBNA, Mercedes-Benz North America.

As will be shown in the next section (3.1), incremental updates to the aftersales data are not available for all aftersales information types. Instead, data is updated using snapshots. Because of the size of these snapshots, a very high bandwidth satellite broadcast will be used to distribute the updated data from the headquarters to the MLCs.

2.3 The PAID Project

The StarNetwork solves the problems of outdated aftersales information and the proliferation of distribution channels mentioned in section 2.1. But a pure online solution introduces new problems: the network and the servers are performance bottlenecks, and may suffer from outages as well.

The PAID project will provide an information distribution platform which will enable the branches to use local databases for the StarNetwork applications additionally to the online access. The most used information is distributed to the world-wide sales organization using the PAID platform. The combination of online and local access offers the best tradeoff between up-to-date content and reliable and fast access.

2.4 Selected Aftersales Information Types

The PAID prototype will focus on a subset of the applications running under StarNetwork. The applications which are currently implemented in the StarNetwork framework are StarParts and StarIdent. The former provides information about spare parts, and will replace the legacy EPC (Electronic Parts Catalogues) application. The latter is a replacement for the FDOK application, which provides information about individual vehicles.

These two applications together with the respective aftersales databases are used for the PAID prototype, and therefore also for the analyzer tool developed within this thesis. The task of the analyzer tool will be to generate minimal updates when changes to these two aftersales databases are made.

Chapter 3

Origination of Aftersales Data

This chapter describes the flow of information for the selected parts of the aftersales database. It explains where the data used within PAID and StarNetwork comes from and why the analyzer tool is necessary.

3.1 Electronic Parts Catalogues (EPC)

As already mentioned in the last chapter, the electronic parts catalogues contain information about which parts are built into the many vehicle variants. These spare parts databases are built up as early as during vehicle construction. Chapter 5 contains a detailed description of the data structures used for storing the catalogues.

Today, three different authoring systems for this spare parts information coexist within MB:

- ELDAS, which is now a legacy system
- DIALOG, the new system for cars
- TINA, the new system for utility vehicles

Along with the introduction of new authoring systems, there were also changes in the way a vehicle was modeled and the spare parts were organized.

It is planned that all information from these authoring systems, and other types of aftersales information not mentioned here, are collected at a central database, called the Mercedes Aftersales Database (MAD). This is done to avoid the proliferation of interfaces between the different authoring and distribution systems. See [MAD98] for details on this.

Figure 3.1 illustrates the current situation for generating and distributing aftersales data at Mercedes-Benz, with focus on the spare parts data.

ELDAS

This is the oldest authoring system, which was initially designed to produce microfiches. It's underlying database is a very old Adabas database system.

The data generated with this system has a semiformal character. Often, rules and connections are described with natural text only. Because of that, this software was characterized as a 'better word-processing software'. Automatic processing of this textual data can be very difficult at some places, including consistency checks.

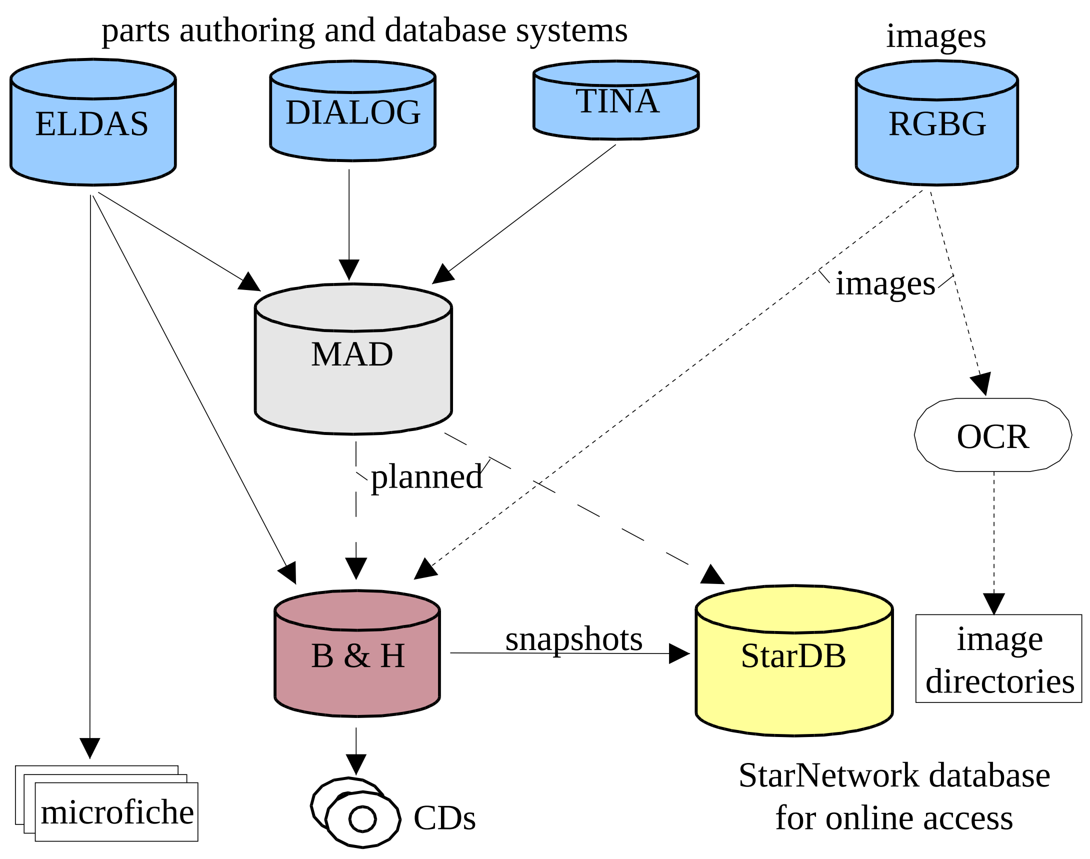

Figure 3.1: Spare parts information authoring and distribution within Mercedes-Benz

The parts catalogues generated by this system are divided up into those for ’models’ and those for ’special versions’. A special version catalog contains only the differences to the model it was built into, which is called the ’positive-negative’ documentation method. This is because a special version catalog contains not only the parts used for the special version component, e.g. an air conditioning system, but also names the parts from the model catalog which are not used (’negative’) whenever that special version is used.

This authoring system makes use of a ’functional decomposition’ of a vehicle. The parts built into a vehicle are divided up into ’construction groups’. Functional decomposition means that parts are grouped together by their function, not by their physical location. Examples for construction groups include electrical wiring, braking system, wheels, doors, ... This decomposition is not very useful when used within an image based part identification process. Parts which are built together will appear on different illustrations, e.g. one will find the brakes on a different picture than the front axle’s parts they are mounted on.

The Electronic Parts Catalogues which are currently distributed on CD and Microfiche to the branches are produced from the export format generated by this system. That export format is called the ’interface ELDAS to EPC’, and is described in all details in [ELD97]. Also, the database schema used within the STARNETWORK database StarDB is entirely based upon that export format. See 5.2.8 for details on how the StarDB relational database schema is generated from this export format.

As I heard recently, it is planned to put the spare parts data coming from Chrysler, a new partner of Daimler-Benz, into this database schema as well.

DIALOG

This is the new system used for creating parts information for passenger cars. It is currently used for the new A-Class, and probably for the next generation of cars under construction while I'm writing this. The paradigm used for documenting vehicles was changed quite a lot. The new method used is called BCT. This is an abbreviation of 'Baureihe-Code-Teil', which means translated 'model line - code - part'. The classification into vehicle models has been dropped. Each part is assigned some codes, which specify under which conditions a part was built into a vehicle. A car is seen as a composition of components and subcomponents. The grouping of components is based on their physical location, as opposed to the functional decomposition into construction groups used by ELDAS.

To make this new scheme compatible with the EPC system, there is a way to simulate the division into models by using some combinations of five codes. Among these are e.g. the codes for the engine and a code describing whether the steering is on the left or right side.

The distinction between models and special versions has been dropped as well. A car is just built up using alternative components. So, this new method is a pure 'positive' documentation, in contrast to the 'positive-negative' documentation used by the legacy ELDAS authoring system.

TINA

This will be the new system used for utility vehicles, e.g. trucks. It uses yet another documentation method called BCS+ which is an abbreviation for 'Baureihe-Code-Stückliste', translated: 'model line - code - part list'. Similar to DIALOG, no model classification is done anymore. But instead of using codes for each part, a code is used for selecting a list of parts. This produces fewer codes that have to be evaluated, but also some redundancy as some parts may be found in more than one list.

RGBG

The images for the parts catalogues are currently originating in a separate system called RGBG (Rechnergestützte Bildgestaltung - computer aided image design), which is located in a different department as well. There is a synchronization gap between the textual data coming from the ELDAS authoring system and these images, as it is not always determinable which illustration set release belongs to which catalog release.

Another problem is that images are not under an explicit version control. Different part catalogues might implicitly refer to different versions of the same image. This gets even worse as the old version of the image is simply overwritten. Some of the images referred to by older catalogues are only available because they were archived by the external distributor Bell&Howell.

Sometimes, coordination of image releases is done via telephone, consider the example of an engineer calling the distributor: "Forget the illustration you got last week, it contains some errors, use the previous version for distribution".

The images coming from RGBG are 'tiff' images which do not contain any information about which parts are shown on the image, and where the parts are located on the image. So an OCR software is used to recognize the part images and image numbers found on a picture. Since the OCR software recognizes only 95% of the images correctly, a special editing software was developed which allows a fast manual correction of errors. Because there are about 100,000 images, manual correction is a challenging task.

Dataflow and distribution

Daimler-Benz engaged an external company, Bell&Howell, for the distribution of aftersales information to the branches. Nowadays, it turned out that the whole knowledge of processing that data and even the quality assurance provisions have been outsourced to that company. As already mentioned earlier, the Electronic Parts Catalogues distributed on CD are based on the legacy ELDAS export format. Currently, 99% of the parts data were generated with the ELDAS authoring system. This might reduce to maybe about 95% in the next few years, but it will continue being the major bulk of data for some time. To avoid generating even more costs within B&H for processing and distributing the parts information, it is planned to convert the new data coming from DIALOG and TINA back to the legacy format defined by the ELDAS to EPC interface description [ELD97]. Even worse, the new vehicle lines must be documented currently with ELDAS in parallel.

The Mercedes Aftersales Database MAD was introduced among many other reasons to solve these problems. The database will be built up in several stages. Stage one, which should be completed this year, includes the automatic conversion of the DIALOG/TINA data back to the ELDAS format. Because of the different hierarchical views mentioned within in the descriptions of the authoring systems above, this is a non-trivial task. The next stage will introduce a common product data model (PDM), which will hopefully bring all aftersales information together in one consistent model. Also, incremental updates will be supported at some time. Another objective was to bring the knowledge of processing and distributing aftersales information back to Mercedes-Benz. Well, seems like 'insourcing' will eventually become tomorrow's vogue word . . .

Another problem is the medium used for distribution of data to the branches and the system used there. There are CD exchangers installed at the dealer's sites which can handle up to seven CD-ROMs. These are shared among the parts catalogues and the vehicle data described in the next section. But, of course, these seven CDs don't suffice anymore to hold all the data. As Mercedes recently tried to remove data belonging to some of the very old vehicle models in order to gain disk space, they faced massive resistance from their dealers. It turned out that the workshops still make a lot of money with those almost-oldtimers.

Currently, the parts data used by StarNetwork is not taken directly from the ELDAS authoring system, but as monthly snapshots from the distributor Bell&Howell. One reason for this is that the ELDAS system is under load doing its daily work, which is for example the export of changed catalogues. Trying to export a major bunch of catalogues additionally makes the system crash. So the exported changed catalogues are gathered since February 1998, but most of the seldom revised older catalogues did not came out of this system yet.

Incremental Updates ?

Small incremental updates to parts catalogues are not supported. The ELDAS interface description mentions an increment containing only the changed construction groups. But that would give rather big increments, up to 100 KB compressed for each construction group where a change occurred. These increments are currently generated very seldom, mostly to do some error fixing after a catalog was released. One must take into account as well, that the bigger a catalog or construction group is, the more often it will be changed!

For part records, there is a tag assigned for marking them as changed or new. But as I discovered, that tag is not consistently used, and can only be used as hint.

Consequence: The Analyzer Tool

Since it is a constraint for the PAID project that the authoring systems are not touched, the only way to get small increments is to regenerate them from the snapshots. This is achieved by the analyzer tool developed within the scope of this thesis.

The main problem in generating a smaller increment between two catalogues is, that the items contained in the catalogues have no absolute identity. Therefore, it is a non-trivial task to find the entries in the old version corresponding to that contained in the new version. This will get worse when the planned conversion of the data coming from the new authoring systems is complete, as then the identity for the items is newly assigned for each snapshot. Some heuristics will have to be used to guess which items in the new snapshot correspond to that contained in the last. The examples found in section 5.2.11 will illustrate the problem.

3.2 Vehicle Data - FDOK

The vehicle database FDOK contains information about every vehicle built since 1986. The database is a legacy IMS/DB hierarchical database. The vehicle data cards stored there are mainly used to determine the right spare parts for a vehicle. The vehicle data cards contain information about the components which were built into a car, for example which engine, transmission, and special versions were used. A detailed description about the data contained here can be found in section 5.3.

Historically, such information was available to the dealers and mechanics in printed form only, and was placed somewhere into the car. Today, vehicle data is available electronically and is fed daily from the factories into a central database.

Figure 3.2 shows the dataflow to and from the FDOK database.

Branches which have 3270 terminals installed can directly access the central FDOK host system. Not all terminals have read-write access. Certain fields in the data cards, e.g. the lock and key numbers, are read-protected as well, for obvious reasons. The vehicle information is distributed to the branches on various media, including the CDs which are shared with the EPC data, microfiche, and even paper. In some countries there is a local mirror database installed. The data is also fed incrementally into the StarNetwork database StarDB.

When storing all vehicles produced since 1986, StarDB would currently have an estimated size of 21 GB. I found a way to reduce this to about 13 GB, which I will present in section 5.3.3.

Feedback and Data Consistency

Since the vehicle data cards should reflect the current configuration of a vehicle, they should be updated whenever relevant modifications are made to the vehicle. The currently only way to do this is via 3270 terminal access, when a workshop has the necessary equipment and network connection. Even then it is not guaranteed that the vehicle data cards will be updated. It is estimated that about 1/3 of the vehicle data cards are in some way inconsistent.

For utility vehicles equipped by an external vendor, this is even worse, as the vehicle data card may be already outdated when the vehicle is sold.

It is wished that the consistency of the data with the actual configuration of the car is improved. It is planned that maybe an automatic feedback from a diagnosis system to FDOK is realized.

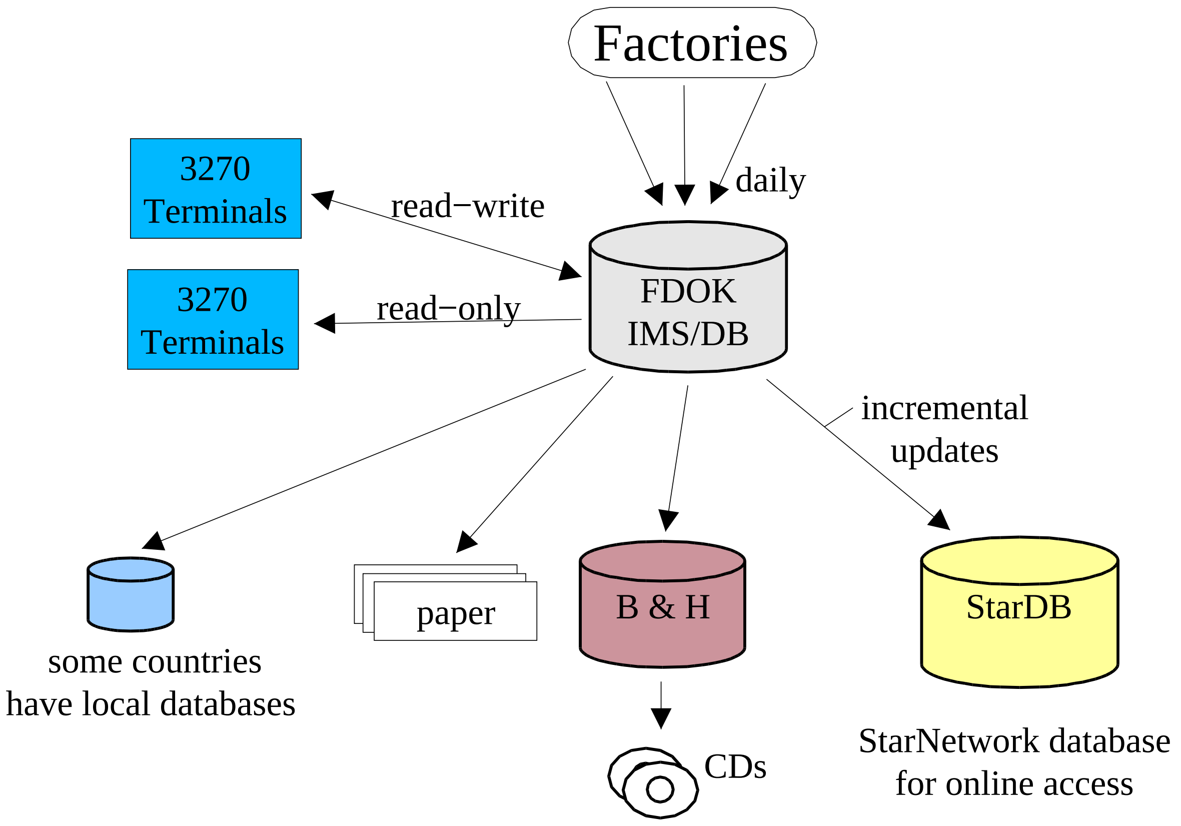

Figure 3.2: Dataflow to and from FDOK database

There is a scrap mark contained in the vehicle data cards in the FDOK database. But this scrap mark is seldom used, an example for this might be a total damage of a vehicle. Since there is no automatic feedback when a vehicle is put out of duty, this database will continue growing by about 1 million passenger cars and 300.000 utility vehicles per year, as long as Mercedes-Benz continues producing vehicles in these amounts.

If a vehicle data card is not available, identifying the configuration of a passenger car manually is possible; for utility vehicles, this is generally impossible. A vehicle data card for a utility vehicle may contain information about more than 1000 different special version components with their respective usage count.

Putting the vehicle data card into the vehicle in electronic form seems to be not a good solution for distribution. The configuration of a vehicle often needs to be determined when the vehicle itself is not available.

Chapter 4

Datamovements within the PAID Network

In this chapter I first describe some specific requirements for a data management subsystem of PAID. In this context, some of the currently available relational database replication techniques are examined. Why did I review these replication systems within this thesis ? The reason is that if one of the replication techniques would meet all essential requirements, the analyzer tool would only need to apply the generated updates on the database, where the replication system would take over and distribute them. It will be shown that the requirements are better matched by an alternative approach, which I call 'Smart Updates'. This chapter gives the answer to the question what output the analyzer tool generates and why.

4.1 Model of Dataflow within PAID Network

Figure 4.1 shows a model of the PAID network, which is a data distribution tree. As this thesis focusses on the data flow and storage, the nodes are shown as database icons. Note that not all of the data is currently stored within a database, e.g. the images for parts catalogues mentioned in the last chapter are stored within a filesystem.

At the top of the data distribution tree is the central StarNetwork database StarDB which holds all of the data. Updates respectively transactions originating there are distributed down the tree to the lowest level, which might be a database within a branch, or even a laptop holding a smaller subset of the database. Daimler-Benz maintains database servers at an intermediate level called 'MLC' for countries or regions. An example for that is MBNA, Mercedes-Benz North America.

The height of the data distribution tree is not fixed, as online accesses are made directly to the central database, and, as well, even a laptop could principally serve as a source node which could transfer updates to e.g. another laptop. I use the names source node and target node here as rolenames for network nodes. As shown on the diagram, a network node on the intermediate level can act as both target and source, depending whether it is seen from the top or bottom of the tree, respectively.

With PAID enabled StarNetwork, an user may access local and remote data simultaneously. Simplified, whenever data is not available locally, it is accessed on the remote database. The benefit of this is that only the most used and critical data has to be installed locally. As already mentioned, it is not within the scope of this thesis to describe all planned functionality of or all requirements for the PAID project.

The connections between the nodes can have very low bandwidth, e.g. modem or ISDN lines,

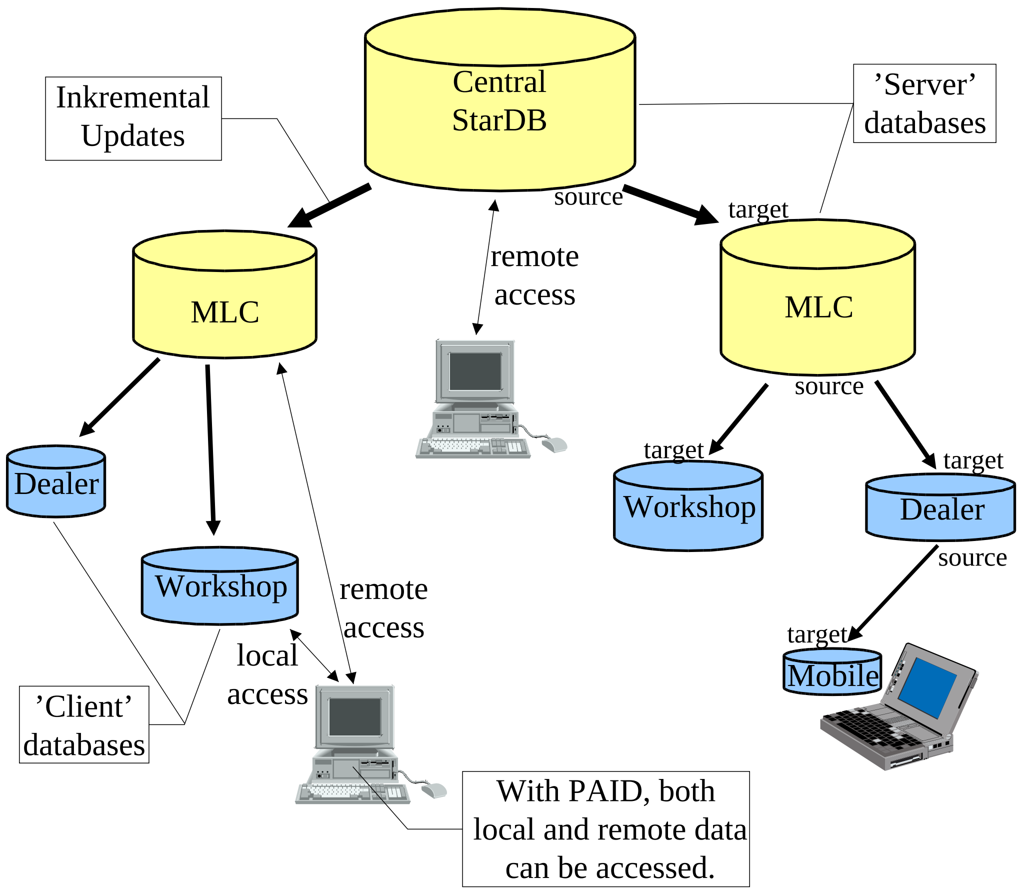

Figure 4.1: Model of PAID network

which are able to transfer between 1-8 KB per second. The sizes of the databases are about 10-30 GB, which is not to be seen as a limit. The total number of network nodes is estimated to about 6000.

Using the same database for all nodes within the PAID network seems to be impossible since the database systems used for the central StarDB and the MLCs have different requirements than those installed e.g. on a laptop. The former are 'server' databases which must be able to handle the high load of queries from users accessing the database online. As opposed to that, the 'client' databases used at smaller branches or on laptops must need zero administration, since a dealer or workshop usually doesn't pay a database administrator. A database installed on a laptop must also have a very small memory footprint. It is a constraint from Daimler-Benz that Oracle is used for the server databases.

4.2 Requirements for Data Movement Techniques

Before looking into the database replication techniques available from major relational database vendors, I will list some of the specific requirements for their usage within the PAID network. Most of the requirements have a general nature, some are specific for relational database replication. The requirements are categorized into essential, non-essential, and 'nice-to-have' requirements.

4.2.1 Essential Requirements

The following are requirements which I consider to be mandatory for a data movement subsystem used for the PAID network. The data movement subsystem is responsible among other things for the distribution of incremental updates.

No full refresh The replication subsystem must never require or automatically initiate a full refresh, which means, loading the whole data over the network. Transferring 20 GB over an ISDN line would take as a minimum a whole month, without considering any overhead.
Two cases must be considered under this constraint: First, it must be possible to setup replication to a database which was already initialized e.g. from CD. Second, even in case of a failure, initiating a full refresh is not an option.

Support for disconnected operation It must be possible for a node to be offline for e.g. 2 month, and, after that period, receive the necessary updates to bring its local data up to date.

Support for message based operation This means, that one can e.g. use a messaging middleware, Email, or simply files as a transport medium for replication. The alternative is that the replication system always requires a direct TCP/IP connection between source and target.
Support for message based operation is necessary for two reasons: The first is the topology of the Daimler-Benz corporate network, where multiple and independently configured firewalls make it difficult to impossible to use uncommon internet protocols and ports. Second, not all branches and nodes are necessarily accessible via a TCP/IP network.
Support for message based operation is a necessary precondition for broad- and multicasting as well: You can broadcast a message, but not a point-to-point protocol.

Support for compression As experience has shown, compression reduces the size of updates to about 3-5%, which makes it at all possible to replicate over a modem line in a reasonable time.[1] Note that this is a requirement for the underlying transport mechanism, not for the replication system. At least as long as the latter doesn't require using its own transport mechanism.

Dynamic data partitioning Replicating to a target node which has only a subset of the source node's data must be supported. The PAID network learns about what data has to be stored on a node, so this subset must be changeable dynamically.

Zero administration Since a dealer selling cars normally does not have any education as database replication administrator, it must be possible to execute all necessary setup and recovery steps automatically.

Vendor independent replication The databases used by PAID should not be limited to one database vendor.

Support for multiple platforms Since PAID itself is platform independent, the replication system should not be limited to only a specific type of platform (e.g. only server platforms/databases, or limited to one platform vendor).

[1]Concrete data amounts can be found in the sections describing the updates to the individual aftersales data categories.

4.2.2 Non-Essential Requirements

These are requirements which the replication subsystem should have, but which are not essential.

One stable queue Since we support disconnected operation, it is not possible to distribute all updates immediately and then forget them. So updates must be stored for some time. Only one queue should be used to store updates which are not yet transmitted. The opposite is a replication system which uses a stable queue for each client, or for each different subset of data which is replicated.
To illustrate this with a realistic example: Consider a change amount of 50 MB per week (uncompressed), and 500 target nodes each holding a slightly different subset which are off-line for a week. You would need 25 GB disk space with a replication system using one stable queue per subset.
I considered this requirement to be non-essential since that disk space might be available at the place where that example is realistic, e.g. at the MLC level.

Efficient use of bandwidth This is especially a requirement for relational database replication: The system should not be limited to transmitting the whole row from a database table when a change occurred therein.
As shown in section 5.2.14 about the size of updates to the EPC database, transmitting only whole records can be very inefficient.

4.2.3 Nice-to-have Requirements

These are requirements which are useful, but one could do without.

Support for broad- and multicast Although PAID claims to be a multicast network, I think it should be sufficient for a replication subsystem to support just the hierarchical distribution of data on a point-to-point base, without using 'real' multicast or broadcast on the network layer. Point-to-point distribution is, of course, supported by all replication techniques.

Changeable source If the source node used normally is not available due to a server failure or network outage, it should be possible to get current data and updates from another node.

4.3 Relational Database Replication Techniques

The Star Network project team also reviewed the replication technologies of three big database vendors: Oracle, Sybase, and IBM. The result was that none of these replication technologies would serve their business needs, which were to support at least the replication to the MLCs over the corporate network. As already mentioned, the data is currently distributed via satellite broadcast.

The information found in the section about the individual replication techniques was taken mostly from the meetings of the StarNetwork team with representatives of these vendors. I added some information I found on the internet as well, including the replication technique from Praxis International.

After shortly describing some aspects of the reviewed replication technologies, and showing in the summary that neither of them meets all requirements, I will present my concept of 'Smart Updates', which meets all of the requirements listed above.

4.3.1 Asynchronous Versus Synchronous Replication

To avoid confusion, I will describe shortly the differences between asynchronous and synchronous replication. This is important, as they are in many ways the opposite of one another. What we are interested in here is asynchronous replication.

Synchronous replication executes changes (transactions to be precise) *simultaneously* on several distributed databases. It therefore often uses the quite prominent two-phase-commit protocol to guarantee that no transaction is ever lost on one of the databases.

Asynchronous replication (which is also called store-and-forward replication) means, that changes are not simultaneously executed on the distributed databases. Instead, changes are distributed to all databases *after* they were applied to at least one database.

Therefore, synchronous replication -

- increases probability of unavailability, because if one database is not ready to commit, the transaction cannot be executed on any other,
- increases response time, as e.g. two-phase-commit needs more than one round-trip between the databases,
- makes conflicting updates impossible.

As opposed to that, asynchronous replication -

- may increase availability, because if the data is stored fully redundant, it is sufficient for one database to be online to allow clients to access the data,
- may decrease response time, as clients can use the fastest accessible database,
- may introduce problems due to conflicting updates, consider two clients changing the logically same field on different databases simultaneously.

More on this topic can be found in [Syb98a].

4.3.2 IBM

Short Overview

IBM supports the following replication scenarios for the DB2 database, which are variations of source and target databases and the possible directions of dataflow between them:

1. Data Dissemination: One source, multiple read-only targets.
2. Data Consolidation: Multiple sources, one consolidating target.
3. Distinct fragments: Multiple sources and targets, but each subset of data has a definite source database, where it can be modified only.
4. Update anywhere: Multiple tables, each of which can be both source and target of changes. This configuration requires resolution of conflicting updates.

The source of replication is specified in SQL, so between source and target the data can be joined, aggregated, derived… . This also implies the replication of views and complex subsets of data specified with SQL. The information about replication sources and subscribers is stored in some control tables, which may reside on a separate control server.

The replication system consists of two components:

Capture, which captures the changes (transactions) made to a database. The changes are read from the transaction logfile. Changes consist of before- and after-images of columns. They are stored in the intermediate ’staging area’.

Apply, which reads the changes from the staging area, and applies them to the target. In ’full refresh’ mode, the Apply program reads the data directly from the source table and copies it to the target.

Whether this is a PUSH or PULL mechanism depends on where the Apply component is located: when running on the target, it’s PULL, when running on the source, it’s PUSH. In each case, Apply uses a direct connection either for getting the changes from the source, or for applying them to the target database. If the Apply program runs in PUSH mode, one update queue is used for each replication target.

IBM is offering the ’DataJoiner’ which allows other vendor’s databases (Sybase, Oracle, Microsoft, ...) to be used as either source or target. DataJoiner makes multiple databases behave as one, and can be integrated into the replication process. The ’DataPropagator NonRelational’ can be used to include non-relational databases. The mentioned staging area serves as a communication platform for the different components.

More information can be found in [IBM98a].

Notes on Suitability for PAID

In case of a failure, and on each cold start of the replication system, a full refresh is made. Also, message based replication is not supported. Source and target databases of other vendor’s are supported, but as opposed to Sybase’s Replication Server, this is not a stand-alone replication system meant for replicating e.g. from an Oracle server to Sybase Anywhere.

4.3.3 Oracle

Short Overview

In an Oracle database used as replication source, changes and transaction are captured with internal triggers. They are stored in a queue of RPC-like requests, and can later be executed on the target databases in parallel. Correct and complete execution is guaranteed with a two-phase-commit equivalent protocol, which requires an online connection.

The supported granularity of data partitioning depends on the scenario which is used for replication:

1. MASTER-MASTER: Several equal databases are connected. Updates and conflict resolution can occur anywhere.
2. UPDATABLE SNAPSHOTS: There is one master site, and several updatable snapshots. There is no communication between the snapshots, so conflict resolution is only required at the consolidating master site. All columns of each table must be replicated, but rows can be selectively replicated.

3. READ-ONLY SNAPSHOTS: In this scenario, there is one master database and several read-only mirror databases. Both rows and columns can be selectively replicated.

In the first two cases, a conflict resolution strategy is required. Such strategies for transactions may be ’first wins’, ’last wins’, and so on. Changes may also be merged, which is useful mostly if different columns are affected by updates to the same row.

More information can be found in [Ora98] and [DDD+94].

Notes on Suitability for PAID

- Experience with Oracle shows that replicating a 10 MB table over an ISDN line is a business for a whole day.
- Replication administration is said to be difficult and error prone.
- A full refresh is done in case of failures.
- Integration of other vendor’s databases is not provided.
- Not all data types used by StarNetwork are supported for replication.

Note on Oracle Light

Oracle Light can be used on mobile computers, and it supports message based replication. This database system requires very little memory. The problem is that it can’t deal with databases of our size.

4.3.4 Sybase

Sybase is promoting it’s concept of openness. This means that APIs to database servers and clients are accessible to third parties. This is also true for the Replication Server, which enables programmers to feed the Replication Server with transaction data. Sybase supports replicating other vendor’s databases, without requiring one of their own database systems as an intermediate stage. They see themselves as a middleware vendor. Sybase offers two different replication systems: The Replication Server and the message based SQL Remote, which is built into Sybase’s Adaptive Server Anywhere.

Short Overview of Replication Server

The information about committed transactions is read from the database log by the so-called ’Log Transfer Manager’. It is then transformed into the ’Log Transfer Language’, which is is fed into the Replication Server at the source database. The data is then sent to the Replication Server at the target database, which applies the changes.

This system is very flexible as you can feed the Replication Server with transaction data from any database system, as long as you can capture it and translate it into the Log Transfer Language.

Including other vendor’s databases into the replication process is well supported. Data from other databases (including legacy platforms as e.g. IMS), can be read by the Replication Server through so called Replication Agents, and can be written to them through gateways.

More detailed information can be found in [Syb98a], [Col93], [GWD94], and [Ren96].

Notes on Suitability for PAID

- Message based replication is not supported.
- Specialized replication administration is required.

Short Overview of SQL Remote

The SQLRemote replication system is integrated into the SQL Anywhere database server, which can be used on mobile computers because of its small memory usage (1MB + typically another 1MB cache). As opposed to Oracle Light, this database system is able to handle our data amounts. SQL Anywhere is designed not to need any administration. SQL Remote is a system which supports message based replication, so changes can be transferred using e.g. email and files.

More information can be found in [Syb98b].

Note on Suitability for PAID

Unfortunately, SQL Remote is not available as a standalone replication system. This is the major reason why it cannot be used for PAID. Adaptive Server Anywhere claims to need zero administration. This might be true for the database system itself, but as mentionend in [Fei97], the replication is not as easy to setup. But without the replication functionality, Adaptive Server Anywhere may still be interesting for the branches and mobile computers, where zero administration of the database itself is a must have.

4.3.5 Praxis International

Short Overview

Praxis International offers the OmniEnterprise suite of products for heterogeneous database replication. Three major products are included in this suite:

OmniReplicator which replicates transactions to other databases.

OmniCopy which does snapshot like transfer of data from a source to a target database.

OmniLoader which extracts a bulk of data into a file which can later be loaded into another database.

Horizontal (columns) and vertical (rows) partitioning of tables selected for replication is supported by these products. Also, data is automatically adapted to heterogeneous target databases, and can be modified or restructured by rules between source and target databases. Replication of changes can be continuous, scheduled at certain times, or event-driven.

The replication rules are setup using a graphical user interface called OmniDirector.

Note on Suitability for PAID

These products currently support only ’server’ databases and platforms, e.g. Oracle and Sybase on Unix and NT, but not smaller database systems used on ’client’ platforms, like for example Sybase SQL Anywhere.

4.3.6 Summary and Rating

The main focus of the mentioned database replication systems is replicating transactions in a secure and reliable way. Opposed to that, PAID needs a very efficient (in terms of network bandwidth) and flexible *data distribution* system.

What I wanted to have (and to put into this thesis) was a table listing all essential requirements and how well they are met by the individual replication systems. Based upon that table, it would have been possible to give a very well founded recommendation for the PAID project. A start of this table is 4.1, but it contains many open questions.

Requirement	DB2	Oracle	Sybase RS	SQL Remote	Omni Replicator
No full refresh	No	No	?	?	?
Disconnected operation	Yes	?	?	Yes	?
Message based	No	No	No	Yes	?
Compression	No (?)	No (?)	No (?)	Yes	No (?)
Dynamic data partitioning	?	?	?	?	No
Zero administration	No	No	No	Yes	No
Vendor independent	Restricted	No	Yes	No	Yes
Multiple platforms	Yes	-	Yes	-	Only 'Server'
One stable queue	?	?	?	?	?
Efficient use of bandwidth	No	No	Yes (?)	Yes	?
Broadcast	No	No	No	No	No (?)

Table 4.1: Requirements for database replication systems.

Unfortunately, I didn't manage to get information about all the requirements for each replication system to complete the table. Requesting information about those requirements from the vendors turned out to be surprisingly difficult. Sybase preferred to not answer on my requests for information. It was also an interesting experience that Praxis International just stopped communications as I asked whether a full refresh was needed initially and after a failure, and if the partitions selected for replication could be changed after replication was started. Well, I guess, if these companies would think that their products meet the requirements, they would answer to avoid loosing the option on selling about 6000 licenses of their replication systems.

Evaluating all available replication techniques by trying them out would go far beyond the time frame and scope of this thesis. At least, the information I got suffices to determine that none of those replication techniques meets all essential requirements. In the next section, I will provide a concept which meets all requirements.

4.4 Replication with 'Smart Updates'

Asynchronous replication distributes transactions respectively the resulting database changes after they were applied to the database. The concept I'm proposing for PAID is to not use a commercially available database replication system for distributing the results of changes, but to distribute the changes themselves. Instead of using a distributed database system, a distributed application framework is created. Figure 4.2 illustrates the differences between the two approaches.

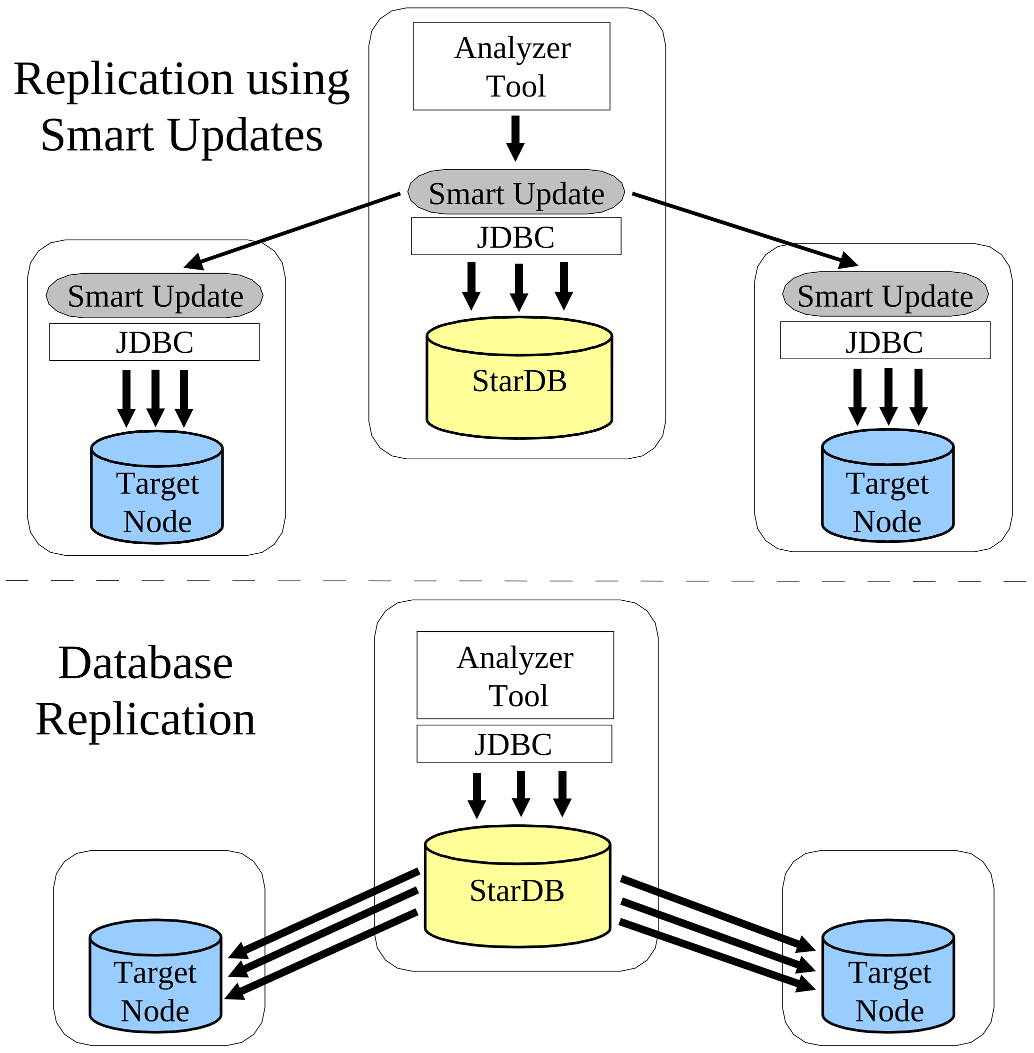

Figure 4.2: Analyzer tool using either database replication or 'Smart Updates'.

Smart Updates are an extension of the 'Command' pattern found in [GHJV94] and [Meh]. An application (or the analyzer tool) would, instead of directly manipulating the local data, encapsulate the request to do this is in an update command. The class diagram 4.3 shows such an update command. The distribution of update commands is done using some kind of message middleware. This is provided by the PAID platform which might use one of the message middleware products for reliably distributing messages that are already available, like e.g. MQSeries [IBM98b] from IBM. JMS [JMS98] is an API recently standardized for using messaging services from Java programs that might be useful.

The Smart Update mechanism can be used easily for distributing data; the analyzer tool developed within this thesis will generate updates in this way. For problem domains where conflicting updates can occur, this gets a little more complicated. But the Smart Update approach enables using semantic conflict resolution more straightforward than on the database level, where inter-table conflicts may be hard or impossible to detect. Since Smart Updates can also be used on a very high abstraction level, they provide the ability to do semantic conflict resolution. As already mentioned, this is a shift from a distributed database system to a distributed application framework. See section about conflict resolution for Smart Updates for some examples and more details on this topic.

Since the local data subset is not necessarily the same for all nodes, the updates are customized to the target, which is an extension of the command pattern described in [Meh]. The description 'smart' refers to this ability of updates to customize themselves to the local data, and to the ability to detect and resolve conflicts in a semantic manner where this might become necessary.

Customization can reduce the network bandwidth used by an update, as parts of the update which affect data not present on the target node can be omitted. If computing time on the source node is rare, this customization of updates can also be done on the target node after the update is received.

Nodes can be added dynamically to the system. Since it should be possible to initialize the local data e.g. from CD, some kind of meta data is used to determine which point in time is reflected by the data. This information is then used to determine which updates are appropriate.

An important property of Smart Updates is the separation of the interface seen by the middleware or framework transporting these updates and the application domain specific implementation which may include some of the business logic to manipulate data. Without this separation, it would be impossible for PAID to provide a reusable framework for handling updates. Despite this, a tight coupling between the semantics of the updates and the distribution platform would result in a bad design anyway.

This concept focusses on the distribution of updates. Further concepts including the functionality to grow or shrink data, initialize databases, and to handle failures efficiently have to be developed within the scope of the PAID project. But most of the implementation used for generating and executing updates can be reused to move data between databases.

The implementation of this concept used within this thesis is located on a very low abstraction level, since the updates are generated automatically. It uses Java as implementation language, the JDBC API to access the databases, and Java's built-in serialization facility to transfer and store the updates. It is described in section 4.4.5.

4.4.1 Meeting the Requirements

No full refresh required Because of the meta data provided to identify the point in time the data reflects, there is no reason for transferring the whole data over the network initially.
In case of a failure[2] occurring during update processing, any appropriate repair or synchronization steps can be triggered. Examples for that include a 'rsync' [TM96] like protocol for efficient resynchronization of databases (if a network is available), or reinitializing the data to an old version e.g. from CD, and applying the necessary updates again.

Support for disconnected operation Updates can be stored for an principally unlimited time until a node is online again. Since they can be stored compressed, and only one stable queue is used (see below), only few disk space is needed.

Support for message based operation This concept is message based, not protocol based.

Support for compression Updates can be compressed before they are stored or transmitted.

Dynamic data partitioning Updates are customized dynamically to the local data subset, so the local data can grow and shrink as PAID learns about which data is needed locally.

Zero administration No replication administration is required on the target nodes.

Vendor independent replication Using JDBC for accessing databases, every database supporting that can be accessed.

Support for multiple platforms Using Java as implementation language makes the updates platform independent.

One stable queue Since updates are customized dynamically, each update needs to be stored only once. An advanced solution could also learn where the updates must be stored for how long, given that at least one node holds the updates until they are (defined to be) not needed anymore.

Efficient use of bandwidth The update commands are generated to be as efficient as necessary. Since very complex operations are practicable for Smart Updates, there is principally no limit for this.

Support for broad- and multicast Since updates can be customized on the target system as well as on the source system, they can be broad- and multicasted.

Changeable source There is no principal need to fix the source node from which updates are received. If e.g. a serial number is used for updates, or information about the version of the data they apply to is provided, the source nodes can be changed on the fly. This is supported by the dependency information of the updates. But this feature has to be supported by PAID respectively the message middleware as well.

4.4.2 Additional Benefits

- This concept is not limited to relational databases. Modifications to files, object database systems, etc. can be distributed as well. Also, a Smart Update could do complex transformations of data between source and target, perhaps replicating from a relational to an object database or filesystem.
- Smart Updates may provide dependency and priority information allowing a reordering of updates e.g. for distributing updates with higher priority first. The sample implementation provides this dependency information for updates. This information allows to determine which updates are valid, and which depend on one another.
- Changes can be generated on a sub-field level. This means, that if e.g. a document is stored in the database, and a change therein occurs, this change could be replicated and applied on the remote database using Smart Updates. This is currently rudimentarily supported for long strings by the analyzer tool.
- Similar changes to redundant data can be encapsulated into one operation. This effectively reduces network bandwidth.
- Smart Updates may provide a notification mechanism when certain data is updated.

[2] Normally, updates distributed over the network are verified to be executable, since they were executed on the source database. If they fail on a target database, the data there is most likely corrupt.

- If the data allows that (as it does for EPC and FDOK), old and new versions can be mixed within the database. The sample implementation given here provides a data model which allows keeping different versions of the data in one database. The benefit is that if the network has a sudden outage, the database can be completed by old data loaded from CD or something equivalent. This can be done without affecting the already up-to-date data present in the database.
- If data is originating at legacy databases, Smart Updates can be used as a backward channel to write updates back to legacy databases. This is easier to implement than on the database level, since the semantics of operations is known and can be translated more straightforward to the legacy data structures.
- A Smart Update might execute complex operations like schema changes.

4.4.3 Outlook: Semantic Conflict Resolution for Smart Updates

Where conflicts cannot be avoided, Smart Updates provide a powerful, reliable, and flexible way of doing conflict detection and resolution. Because they are not limited to the results of data manipulations, they can do some smarter conflict resolution steps than normal database replication can do.

There are many different conflict resolution strategies available, [Syb98a] contains many examples for that. All of the database replication techniques offer some automatic conflict resolution strategies, which are of the kind ’first transaction wins’, ’last wins’, and so on.

Alternatively to these automatic conflict resolution strategies, Smart Updates allow easier implementation of semantic conflict resolution than implementing that on the database level. Since this is not the focus of this thesis, I will only provide some examples and ideas for this here. The following scenario is an example for resolving conflicting updates to the vehicle database:

- An air condition is built into a car. A workshop adds the air condition code to the list of special versions built into that car. But the update is not propagated up to the central database, since the workshop is currently offline.
- Soon after that, the car has a failure. Another workshop, which repairs that failure, notices that the car has a built-in air condition, but that the corresponding code is missing in the database. It also adds that code to the list of built-in special versions.
- The update that reaches the conflict resolution site later will notice that the code for the air condition can’t be added to the list, since it is already there. It simply drops itself as it discovers that it is redundant.

Another, more advanced scenario is the following:

- Again, a mechanic does an update to a vehicle data card. He enters the code of a new special version into the data card. He presses ’Ok’, and gets the message “Your update has been preliminarily accepted”.
- The local database is temporarily updated, but the original version of the vehicle data card is preserved.
- The update makes its way up to the central database. Unfortunately, it discovers there that the vehicle data card has changed in the meantime, and that it doesn’t know how to do conflict resolution in this case.

- A notification is send back to the workshop where the conflicting update was initiated. The next time the mechanic logs in, a message is displayed saying “Your update ... has been rejected. Click ’Ok’ to try again.” After clicking ’Ok’, the application displays the new version of the vehicle data card along with the update which caused a conflict and the reason therefore.

- The mechanic can now resolve the conflict manually.

Of course, an application which wants to provide such means of conflict resolution must be designed from the start as a distributed application supporting disconnected operations. Scenarios like those above are only realizable if there is a small number of valid operations which might cause conflicts. This concept is e.g. feasible for the vehicle data card database, where the data structures are very simple.

Wherever possible, it is preferable to avoid conflicting updates when designing a distributed database system or distributed application. An example for this is the Usenet, where all people may write to a discussion group, but can never conflict with each other.

It is important to consider that a manageable conflict detection and resolution requires the introduction of a definite clearing house. This doesn’t need to be exactly one node, but there should be only one responsible node for each data partition. When at some time the idea of the central after-sales data maintenance and distribution is dropped, it might for example be chosen that every MLC database does conflict resolution for its area specific data. But if every node in the network could do conflict resolution, this will soon get unmanageable. Using e.g. database replication within an update anywhere scenario, every site may get the updates in a different order, and therefore might undertake different steps to resolve conflicts. If, in such a scenario, the conflict resolution strategies are not carefully chosen, it might even happen that the system never reaches a stable state after some conflicting updates, as every site may see the conflict resolution actions of the others as new conflicts. Some more information about this topic can be found in [Syb98a].

The following are possible variations of extending Smart Updates with the ability to do conflict resolution:

- an update request includes rules for detection of conflicts. These rules may differ from one type of update to another (even for the same set of data) and are therefore highly customizable.
  Most of these rules would have to be implemented anyway, as an application normally checks whether an update to the data is valid. These rules could be reused to do conflict resolution.

- instead of just being accepted or rejected, an update may adapt itself to a new situation, where it would normally be rejected in its original shape.

- an update may be only temporarily applied. The change is laid like a ’foil’ over the ’picture’ of consistent data. Simultaneously the change is sent to the ’clearing house’, where it will approve, modify or reject itself.
  For some types of data, this can be implemented very easily. For example, in the vehicle data card database, a tag might be included for each vehicle data card which tells that it was modified locally. The modified vehicle data card is stored at another location in the database. As soon as the modification is accepted, and the update to the original vehicle data card is received, this copy is dropped.

- an update which is rejected may present this fact to the user which caused the conflicting update in the most suitable way by using the same application that caused the conflict.

As already mentioned, these are only some ideas of how conflict resolution can be done. Implementation of a fully featured conflict resolving update can be very challenging, depending on the possible updates. But, implementing these rules on the database level is even more challenging. The standard rules, e.g. ’first wins’, ’last wins’, ..., can also be very easily implemented for Smart Updates, if a smarter conflict resolution is not needed. Where conflicts happen very seldom, this might be the preferable way to save implementation costs.

4.4.4 Conclusion and Decision

The only limits for Smart Updates are the complexity and implementation costs. Using Smart Updates, nearly everything is possible, as long as it can be implemented at reasonable costs. But, if a middleware framework for distributing them is available, the implementation costs to get the same functionality as for database replication are very low. Such a framework could also provide an intermediate JDBC driver that replicates all transactions to the database as automatically created updates. This is equivalent to database replication, except that the latter normally reads the changes from the database logfile.

I don’t believe that there will soon be database replication techniques available that match the requirements of the PAID project. Replicating Gigabytes through 1 KB/s modem lines wasn’t advertised by any replication technology vendor.

Another important point I had to take into consideration was the approach which would be used to solve the problem that the data amounts were too big to be replicated through the network. One could redesign the data structures in a way that updates to the database would be smaller. But I rejected from doing that because I cannot assume that this is possible for all current and future databases which one might want to distribute using a PAID like distribution platform. An example for this is the WIS database, where very complex data reorganizations and relocations are made between releases. Requiring the data structures to be designed such that only small updates are generated is often not feasible, since data often originates from legacy databases.

So I chose to implement the analyzer tool to generate Smart Updates. Since I don’t know what particular way of distributing updates will be chosen for the PAID project, I provide only a very rudimentary implementation of updates within this thesis.

As I will show in the chapter about the FDOK aftersales data, Smart Updates can reduce the size of updates to this database in a way that cannot be done using database replication. I also think that the idea of replicating to any kind of database which is accessible through JDBC is very promising.

4.4.5 Object Design of Smart Updates

I present the toplevel implementation and interfaces of the Smart Updates now, because it serves as an example which should help demonstrating the conceptual ideas. But the design introduced here should not be taken as a reference, since most functionality was not or only rudimentarily implemented. The concept of data subsets introduced here is later used for presenting the subdivision of the EPC and FDOK databases in chapter 5.

Within the scope of PAID, most likely a more general model of data and of the dependencies therein will have to be developed. This will be necessary to handle not only the updates but also the valid local configurations for initializing and modifying the subset of data locally mirrored.

The class diagram 4.3 shows the main interface ’Update’, which represents a Smart Update, and the other classes involved with the management of updates.

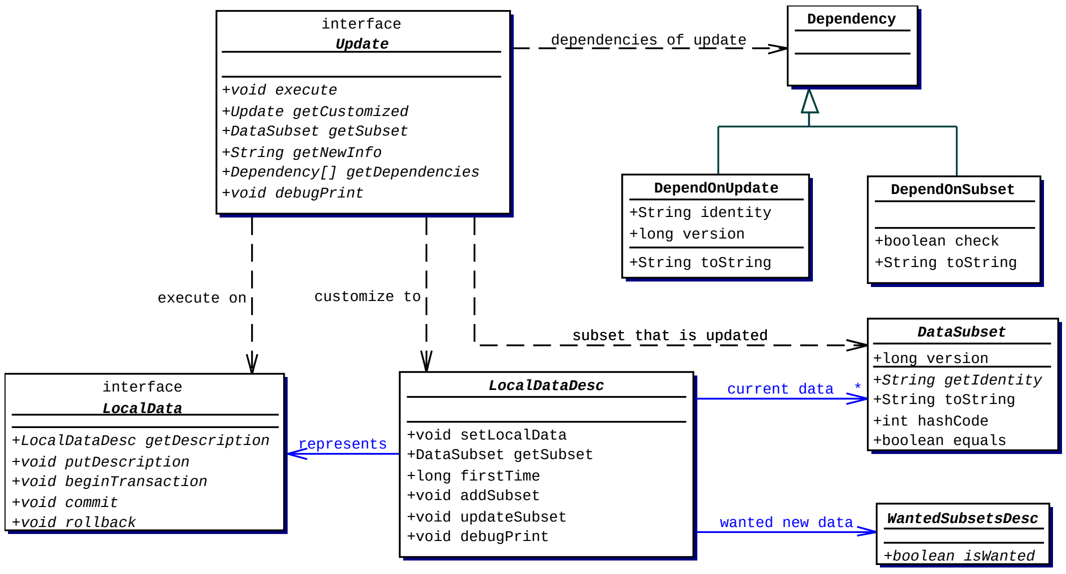

Figure 4.3: Smart Update and involved classes

The following sections will explain the methods which must be implemented by a Smart Update and the classes shown on the diagram. Not all methods are described here; a complete reference can be found in the javadoc documentation accompanying this thesis.

Methods of Update

This is an overview of the methods implemented by an update; It contains forward references to the next sections describing the involved objects.

execute This is the main method of the update, since it applies the update to the local data. A reference to this local data is given to this method as a LocalData object.

getCustomized Creates a new update which is customized to the local data. A description of this data is given as parameter to this method (as LocalDataDesc object).

getSubset Returns a DataSubset object describing the new version of the data subset which is updated. This object also uniquely identifies the update.

getNewInfo If the update introduces a new data subset, this method returns a string describing the new data subset. This string can then be used by WantedSubsetsDesc to decide whether this subset is wanted or not.

getDependencies Describes the dependencies of an update.

debugPrint Prints debugging information on the standard error output stream.

The message sequence chart 6.4 found in section 6.2.5 describing the implementation of the apply subsystem of the analyzer tool will illustrate how these methods are used.

Description of local data (LocalDataDesc) - DataSubsets

The class LocalDataDesc provides a description of the data that is installed locally. Data is divided up into DataSubsets, each having a name (identity) and a current version, which is represented by a timestamp. The designation 'data subset' is quite insignificant; this is because it is a very general superclass which may e.g. reflect a catalog in the problem domain EPC, but a combination of model line and area within FDOK.

As I searched for a suitable representation of data subsets, I discovered that complexity is reduced very much if data subsets are modeled in a stable way, which means that data never changes the subset it belongs to. If it would do that, it might happen that its new data subset has an incompatible version or isn't installed locally. These conditions would have to be checked for and handled whenever data is updated. The following are further requirements for well modeling data subsets for the respective application domains:

- Since data is accessed both locally and remotely, the subsets should be chosen in way that queries can be *either* answered by the local data *or* remotely.
- If there are only few dependencies between subsets, management of updates and data is easier.
- If subsets are chosen in a way that different versions can be mixed, it is possible for example in case of a network outage to install the now missing remote data locally as an older version from CD. This requirement is also a precondition for enabling reordering of updates.

The way subsets are chosen for the two reference aftersales databases gives an example for data subsets matching these criteria. Since there might be updates that introduce new data subsets, the description of local data is associated with a description of which new data subsets a target node wishes to get (WantedSubsetsDesc).

Note that there is always a way to implement this concept consistently for a specific problem domain, because it is always possible to model the whole data as one subset.

Dependencies

An update can have dependencies. The most common is that the data subset the update applies to has the appropriate version. An update may also request that another data subset has a defined version, which applies only if the data subsets are installed locally. These two depedencies are represented by a DependOnSubset dependency.

There is also a way to link updates together. It is possible for two updates to depend on one another, which is expressed by DependOnUpdate. This indicates that the updates should be applied simultaneously, e.g. within one transaction.

Other dependencies might be introduced for specific problem domains.

LocalData

LocalData is a reference to the local data. It provides the method getDescription() to read the description of the local data (LocalDataDesc) from the database, and putDescription() to write it back.

The class also provides a generic transaction handling capability. Before updates are applied or the local data description is modified, beginTransaction() is called. Then all updates which depend on one another are executed. The description of the local data is updated as well. After all this is done, commit() is called. This ensures that data never has an inconsistent state, and that the description of the local data is consistent with the local data.

About Consistency

There are certain consistency constraints for the proposed mechanisms to work reliably. The transaction concept offered by the databases is used to prevent data corruption because of errors or breakdowns. But certain things have to be kept in mind when designing the system. One is that the subset of data installed must not be changed while already customized updates are transmitted or in progress of execution. Also, when that local data subset grows or shrinks, the corresponding local data description must be adapted within the scope of the same transaction used for that purpose.

This concepts are not yet implemented fully, since I only provide a proof-of-concept implementation and I don't know which kind of transaction service will be available within PAID.

4.4.6 Needed Improvements

The way of representing data subsets has to be rethought. For example, the concept of representing and identifying data subsets and new data subsets by strings was driven by the way the updates are currently stored in a database queue (table). I wanted to make that information available in human readable form outside the update object. The chapter about the analyzer tool provides more details on this. I also used this representation of data subsets and versions to produce statistics to find the optimal way of producing updates.

It might be useful if an update is not limited to updating only one DataSubset. Complex updates like schema changes affect more than one DataSubset. Some more clean and general model of data and meta data will have to be developed for representing the data subsets, their version, and the consistency constraints.

4.5 Alternative Approaches

4.5.1 Remote Synchronization

As an alternative approach, one could use a remote synchronization algorithm to bring two databases up-to-date. 'Rsync' [TM96] implements this for files; in [BR97] a similar but more advanced technique for doing remote synchronization of databases is described. There is a Java tool set available which performs synchronization in a very simple way ([Shaf]).

Using synchronization, no state information would have to be maintained, which is obviously a very big advantage. Adding nodes and shrinking or growing of a node's local data is uncomplicated. No analyzer tool would be needed as well, since no information about changes between releases is necessary.

The idea behind synchronization is to compare two remote data sets using a certain protocol. The rsync approach e.g. transmits checksums to identify differing data blocks in files.

The general problem with this kind of approach is that when the size of these blocks is small, then there will be very many blocks which will have to be negotiated. If the identified blocks are big, then too much unchanged data will be transmitted. So one would start to negotiate bigger chunks of data, and then proceed only in the differing ones to find small changed subsets. This works very fine if the changes are condensed, since then only a few differing blocks are found. But using this approach is generally inefficient if the changes are widespread all over the database in an unpredictable way, which is the case e.g. for the parts catalogues.

This way of synchronizing databases would never be as efficient in terms of network usage as generating the necessary updates locally and distributing them. I guess that it might not even be usable for synchronizing gigabytes of data through modem lines, especially when a node was offline for some time and the data has changed at many different places. I think this is obvious, but in lack of a test environment, I couldn't verify it.

Since it is a protocol based approach, a direct connection between the source and the target node is required. This kind of synchronization can hardly be done effectively in a message based environment, without using too big units for synchronization. Broadcasting and multicasting is of course impossible using this method.

Another problem with this approach is the computing time on the source node, when a few hundred target nodes would want to synchronize themselves at the same time.

An approach of this kind is although very interesting for doing an effective resynchronization after a failure. But it would then had to be tested that the resynchronization is done in a reasonable time for the most likely cases. While doing synchronization, an estimate has to be made how long it will take. This is necessary because synchronization might produce worse results than transmitting the whole data. It must be kept in mind that transmitting 5% of the aftersales database through an ISDN line takes a whole day.

4.5.2 Timestamp Replication

A very simple approach to distribute database changes is to assign timestamps to each record. For each data subset, a target node would receive all records that have been changed after the newest record in that subset. But this technique has some drawbacks:

- Moving of records cannot be captured that way. A record is said to be moved when its primary key values are changed. The effect of this is that you will eventually find no record or (logically) another record at the place the original record was.
- Similar updates to many records are not replicated efficiently using this method, since the results of the changes are not necessarily similar. Similar updates can be compressed very well, but not the different results.
- For some problem domains, a record or even a field may be a too big unit to be transmitted. An example for this is a database storing documents that were generated from a template. If that template would be changed, all documents would have to be retransmitted. Equipping all possible data subsets with timestamps might not be realizable, depending on the nature of the data.

The first two points apply to the EPC database, at which the first one is worse, since the design of this database results in such moves between releases. The sections about timestamp replication found in the respective sections for EPC and FDOK in the next chapter will explain how this approach could although be used alternatively to Smart Updates.

Chapter 5

The Aftersales Data

In this chapter, the data structures used for the relational aftersales database StarDB are described. The two database schemata are named EPC and FDOK in this thesis after their legacy predecessors. The EPC data is used by the StarNetwork application StarParts, which will be used at the branches to identify the parts built into vehicles. The FDOK data is used to determine the configuration of an individual vehicle, for example which aggregate variants and special versions are built into it. The corresponding StarNetwork tool is called StarIdent. The database schemata within StarDB are named after the new terms, “Parts” and “Ident”, and can be found in appendix A.5 and B.1, respectively.

I preferred to use ’EPC’ and ’FDOK’ to avoid confusion since I often have to use the words part, parts, identifier, and identify.

It is assumed that the reader is familiar enough with relational databases to know what a table is, how data is stored therein, and what a primary key is.

5.1 About Class Diagrams Describing Legacy Data

The class diagrams contained in this chapter are not comparable with class diagrams made from scratch for designing a new system. They are reverse engineered from a relational database implementation designed for storing legacy non-relational data. They are to be seen as no more than a utility to show the objects contained in the databases and the associations between them.

The schemata for the databases were designed to allow a straight conversion of data coming from some legacy file formats. See section 5.2.8 for details on how this is done for EPC. Because of the legacy data structures, neither entity-relationship diagrams nor class diagrams could be used straight on for modeling the data structures as they are implemented for the StarNetwork databases now. Some compromise had to be found between drawing meaningful and correct class diagrams and staying close to the real implementation. The textual descriptions found in the sections for the individual objects respectively tables will illustrate this problem. To stay as close as possible to the real implementation and to avoid introducing even more classes, I tried to establish a one-to-one mapping between tables and classes.

Another problem I had to deal with was the mixture of languages. The tables and fields used in the database are german or abbreviations of german words. So I couldn’t use the names of the fields and tables from the database within the diagrams.

So the following conventions are used for the class diagrams within this chapter:

- The stereotype fields are (mis-)used to name the table the objects are stored in, and/or the fields their identity is represented with. Unfortunately, this was the only way to place notes inline.

- The attributes shown in the diagram do not reflect the identifiers or types used within the database. They just give some hint of what data is stored within an object. I also omitted some fields which were redundant or insignificant. Putting all attributes from the EPC or FDOK database schemata into the diagrams would render them unreadable.
- The CASE tool TogetherJ does not correctly draw qualifiers. As they are quite confusing the way they are drawn, I omitted them completely.

Since only data is modeled, the classes do not define any operations.

5.2 EPC - StarParts

5.2.1 Utilization of Parts Information

The Electronic Parts Catalogues are used to identify which parts are built into a specific vehicle.

This information is needed at the branches for example to sell parts directly to the customer, or to order parts needed for a repair. It is also used internally, for example to support the authoring of service documents.

With the introduction of StarNetwork this identification process is totally image based. The user selects a part on an illustration and immediately sees the corresponding part record (=part description).

Unfortunately, the selection of the proper part is not fully automatically. At least not with the data coming from the legacy ELDAS authoring system. Some information must still be manually gathered and reviewed from footnotes and so-called supplementary texts. Therefore, the part identification process can still be error-prone.

5.2.2 Overview

Electronic parts catalogues contain images and parts information. Images are stored within a directory tree, parts information in a relational database schema consisting of 26 tables. Appendix A.1 shows a complete list of all tables along with a short description.

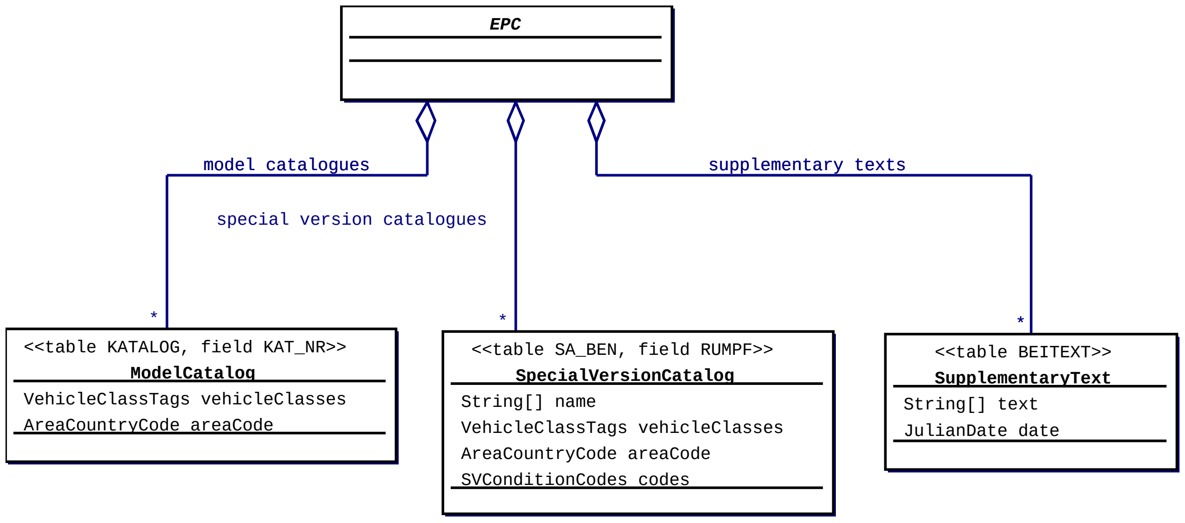

Figure 5.1: Overview of Electronic Parts Catalogues database

As shown in the figure 5.1, the EPC database can be divided into three subsets:

- model catalogues (described in section 5.2.4).
- special version catalogues (described in section 5.2.5)
- supplementary texts (described in section 5.2.7)

These subsets are loaded independently into the database. The next sections describe these three subsets in detail. Additional information is found in the appendix: A.5 contains the SQL schema definition, and A.4 may be very helpful if you don't understand german as it contains a reference and translation for all field names found in the database schema.

5.2.3 Language Support

The parts database supports six languages, listed in the following table (5.1).

language	suffix	value
neutral	not used	0
English	_E	1
French	_F	2
Spanish	_S	3
Portuguese	_P	4
Italian	_I	5
German	_D	7

Table 5.1: Languages supported by EPC.

In the class diagrams, I consistently used string arrays (String[]) to represent the fact that a description, name, or title is available in several languages. A two dimensional array (String[][]) means that there are several lines of language dependent text. Within the database, each line of text is stored in a separate row of the respective table.

Three different methods are used to represent language specific data in the parts database:

- A separate column is used for each language, using a suffix for each language (see table 5.1 above). This is the most common method. The suffix '_X' is used for those names within this thesis.
- A special column which identifies the language. This is used only for footnotes and special version catalog names.
- One table for each language. This method is only used for image group titles. The table names have the same suffixes as used for the columns.

The different methods to store language specific data result from the legacy input data files. If multiple lines were used there to store the text, then multiple rows are used in the database as well.

5.2.4 Model Catalogues

Model catalogues contain the images and parts data for the regular models, without any special version equipment. The class diagram 5.2 shows the objects and associations contained in the regular model catalogues. In the following sections the objects, attributes, and associations shown on the diagram and their representation within the database schema are explained. All data contained in a model catalog except the information about images is converted and loaded into the database from a single file.

Model Catalogues (table KATALOG)

A model catalogue is a closed unit containing part records, footnotes, image references, and model information for the main vehicle or aggregate models. It can contain data for up to 22 vehicle models. This is a historical limitation, as that was the number of columns which could be shown on a microfiche. A model catalog is identified by a three character id (KAT_NR). A column with this name is found in the primary key of all tables used for the model catalogues.

Two attributes specify which vehicle classes a catalog is valid for (column SORT_KLASSE), and for which area respectively country a catalog is used (column BER_CODE). Appendix A.2 lists all valid area codes and A.3 all vehicle class identifiers.

As the part records in a model catalog document all models in parallel, mostly similar models are grouped together in one catalog. This saves space, as only one part record has to be used for a part which is valid for all models. Wherever parts are differently used among models, several part records have to be used. This is because one part record describes for exactly one part the models it is built into. See section about part records for details.

Models (table BAUMUSTER)

A model, which can either be a chassis model or an aggregate model, is identified within a catalog by two three-digit numbers: BR (model line, for german: Baureihe) and BM (model, for german: Baumuster).

The following table (5.2) gives some examples of models.

catalog	BR	BM	model type	name
45Z	208	465	chassis	CLK 320 USA
46C	208	465	chassis	CLK 320 JAPAN
35G	904	323	chassis	SPRINTER 408 D
19N	111	974	motor	M 111 E23 USA
19L	111	974	motor	M 111 E23 JAPAN
07E	717	450	transmission	GL 76/30 A-5
01J	730	408	front axle	VL 0/6 CE - 1,6/RS3550
01Y	763	000	steering	LZS 1

Table 5.2: Examples of vehicle and aggregate models.

A type field is used to distinguish between aggregate and chassis models. For chassis models, all aggregate models which can be built into it are found in the table FGST_AGG. Vice versa, for an

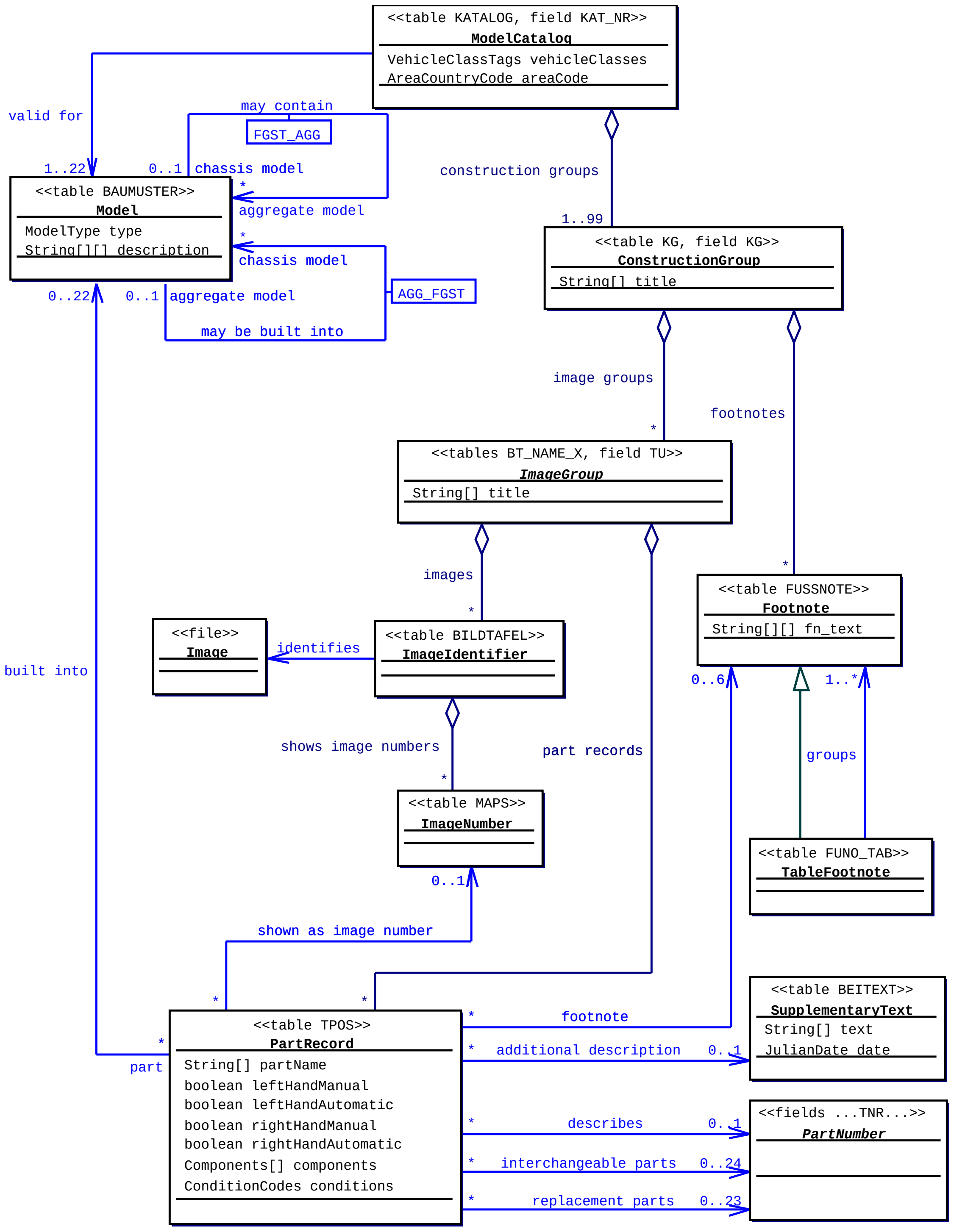

Figure 5.2: Pseudo class diagram of model catalog objects

aggregate model, all chassis models which it can be used with are listed in AGG_FGST. Note that this is redundant and an artifact of the way the data is divided up into catalogues.

The data for one model extends over two or more rows in the database table: In the first row, the type of the model and the column containing a language neutral description (VERK_BEZ) are valid. The following rows contain one or more lines of descriptions, where the columns for the respective languages (BESCHR_X) are valid.

Construction Groups (table KG)

The image references, parts, and footnotes are organized in constructions groups. They are shown in the diagram as 'part-of' a construction group. The two-digit construction group identifier 'KG' is found in the primary key of all tables used to store that objects.

The following table (5.3) shows some examples:

KG	title (english)
20	ENGINE COOLING SYSTEM
25	CLUTCH
40	WHEELS
54	ELECTRICAL EQUIPMENT AND INSTRUMENTS
72	DOORS
82	ELECTRICAL SYSTEM
91	FRONT SEATS

Table 5.3: Examples of construction groups.

Image Groups (TU)

Part records and images are divided up into image groups, which is a rather arbitrary subdivision. It is just a range of part records shown on typically between one and five illustrations. An image group is identified by the 3-digit field TU (parts accumulation, for german: Teileumfang). Each image group is assigned a title, which is stored in a separate table for each language (the BT_NAME_X tables). The subdivision into image groups is not always stable among releases of model catalogues. Images and parts are sometimes reorganized into new image groups.

Images and Image Numbers

An image shown in the parts catalogues is typically a construction diagram showing some parts. The attributes from table BILDTAFEL are used solely to construct the filename for the images found on disk. A release date is used to distinguish between several releases of an image. This release date is necessary because it happens that different model catalogues refer to distinct release dates of the same image.

On an image, several parts are shown with numbers near them. The table MAPS tells which image numbers (field BILD_NR) are shown to the user on an image. Within an image group, an image number is always used for the same part, and may be shown redundantly on more than one image.

Image files are not stored within the database. The image files contain additional information which tells the StarParts application about enclosing polygons of the part shown for an image number. This makes it possible to highlight the parts when they are selected by the user.

Part Numbers.

Part numbers are unique and stable identifiers assigned to parts. Whenever a column contains the string ’TNR’ (Teilenummer), it contains one or more part numbers. Most part numbers contain the model line (BR) they are used for as a prefix.

Part Records (table TPOS)

This is the main object in the parts catalogues. Most records contain a valid part number in the field TNR, but there are also some note records with the special part number ’XX...XX’.

A part record describes under which conditions its part was used within a specific vehicle or aggregate. Some conditions are only available as natural text linked to the part record as footnotes or so-called supplementary text.

Some conditions can be evaluated automatically. Examples for that are some flags which indicate whether the part is being used with a vehicle which has left or right hand drive and a manual or automatic transmission. There is also a field called CODE_BDG which contains some special condition code numbers. For utility vehicles, the fields starting with ’KP_’ list some component main numbers and their so-called stroke versions the part is used with.

A part record also tells in which amount the named part number was built into the vehicle models the catalog is valid for. This association was carefully hidden: The string attribute ALLE_MENGEN lists the amount as 22 three-digit numbers. The table for the respective models (BAUMUSTER) contains a column POS_NR which identifies the starting position within this string for each model. You can think of this as a matrix, where models are the columns and part records the rows.

A part record is uniquely identified by the catalog id, the construction group, and a simple running number (LFD_NR). That running numbers sometimes change between releases of a model catalog, but are mostly stable. This will change when the planned conversion from the new parts authoring systems DIALOG and TINA is complete. Then the running numbers will be assigned temporarily for each snapshot conversion.

Sometimes a part is being replaced by others, for example if there were new versions of the part available and the original part is not produced anymore. Because one might search for the original part number, the part record is not deleted, but instead a flag is set and another field lists the replacement part number(s).

Another flag is set if there are equivalent parts which are interchangeable with the part a record is for. Then an attribute lists the part numbers and respective amounts which can be used instead of this part. This happens for example when equivalent parts are provided by more than one external vendor.

Section 5.2.11 about the changes between the EPC releases contains some examples for part records.

Selecting Valid Images for Part Records

The part records contain the references to the image numbers used within an image group. After detecting which image numbers are referenced by valid part records, the table MAPS is used for the sole purpose of detecting which images in the particular image group show the valid parts, as this table establishes an association between images and image numbers.

Footnotes (tables FUSSNOTE and FUNO_TAB)

A footnote consists of one or more lines of text. Each line is either language neutral or available in all supported languages. A footnote is identified within a construction group by a unique number (column FN_NR). The line number within a footnote was split among two columns called FN_FOLGE and FN_ZEILE, which is an artifact from the conversion process. The language of a line is determined by a separate column called FN_SPRACHE (footnote language).

Some footnotes are grouped together to form a table footnote. This is often used for color tables, which are referenced when a part is available in several different colors. These table footnotes are stored in a separate database table called FUNO_TAB. The column BLOCK_ZAEHLER in that table is then used to group footnotes. If one of the footnotes numbers contained in a table footnote is referenced from a part record, all footnote lines with the same value in this column are shown.

A consequence of the use of two separate database tables is that one might have to look into both to find a referenced footnote number.

5.2.5 Special Version Catalogues

Special version catalogues contain images and parts data for special versions. The examples for special versions contained in the corresponding section below give an idea of what a special version can be. Figure 5.3 shows a class diagram of the objects contained in a special version catalog and their associations. Special version catalog data is loaded separately from model catalog data into the database. All data for a special version except the information about images originates from a single file.

Special versions are always relative to some specific vehicle models. The catalogues for them will only contain the differences to these models. Section 5.2.6 will illustrate the connections between special versions and regular models.

Special version catalogues are in many ways very similar to model catalogues, but the hierarchical view is totally different. Since they list only parts contained in a few variants of a special version and not all parts from a complete vehicle, they are much smaller than model catalogues.

Special Version Catalogues (table SA_BEN)

A special version catalog is a closed unit containing part records, footnotes, image references, and information about sub-variants of a specific special version called stroke versions. A special version is identified by the six-place main number called RUMPF. A column with this name belongs to the primary key of all tables containing special version data.

The table SA_BEN contains two attributes which specify which vehicle classes a catalog is valid for, and for which area respectively country it is used. These are the same attributes (SORT_KLASSE respectively BER_CODE) as used for the model catalogues. Appendices A.2 and A.3 list all valid area codes and vehicle class identifiers.

Additionally, there is a condition field called CODE_BDG, which specifies some conditions for how the special version may be used. Another column contains the name used for the special version, using one line respectively database table row for each language. The following table (5.4) gives some examples of special versions.

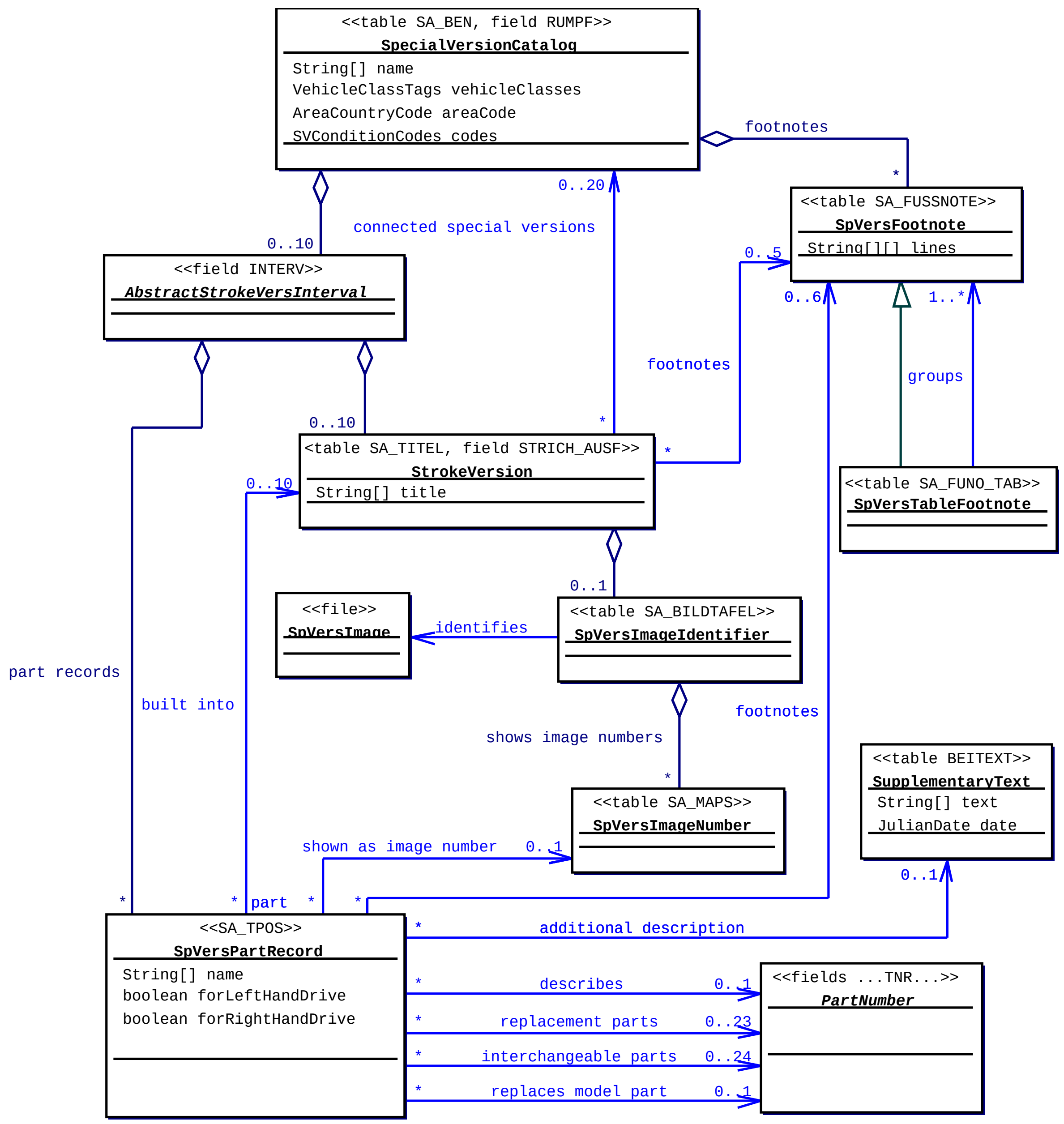

Figure 5.3: Pseudo class diagram of special version catalog objects

Stroke Versions

A stroke version is a sub-variant of a special version. It is identified by a 2 digit number, so there are up to 99 stroke versions per special version main number. Some stroke versions exclude one another, others must be combined. Therefore, for fully specifying which special version variants are used within a vehicle, more than one stroke version might have to be given for a special version main number. All database tables which contain stroke version specific information have a field named STRICH_AUSF (stroke version) within their primary key.

The table SA_TITEL contains some lines of title for each stroke version and optional references

main number	special version name
12318	ENGINE PARTS W/CRUISE CONTROL (TEMPOMAT)
12439	AIR POLLUTION CONTROL
10987	MAXIMUM SPEED LABEL FOR USE WITH M & S TIRES
11341	SAFETY-BELT/SPEED WARNING SYSTEM; FOR GULF STATES ONLY
086709	WOODEN FLOOR
086768	CHASSIS PARTS FOR LONGER WHEELBASE
086746	FIRE EXTINGUISHER
500004	SV ENGINE REAR SUPPORT

Table 5.4: Examples of special versions.

to up to five footnotes. Also, there is a field which specifies up to 20 special version main numbers which are in some way connected to a stroke version. An example for this is a stroke version for an antenna connected to the special version main numbers of various radio models.

The following table (5.5) gives an example of stroke versions for the rear engine special version shown in the last table. It is also an example for how multi-line text is stored within a database table.

main no.	stroke vers.	line	stroke version title
500004	01	0	ONLY WITH MECHANICAL TRANSMISSION
500004	02	0	ONLY USED W/AUTOMATIC 4-SPEED
500004	02	1	TRANSMISSION
500004	03	0	ONLY USED W/AUTOMATIC 5-SPEED
500004	03	1	TRANSMISSION
500004	04	0	WITH AUTOMATIC TRANSMISSION, MAIL VEHICLES
500004	05	0	ONLY WITH MANUAL TRANSMISSION POOR ROAD
500004	05	1	VERSION, CODE Z11
500004	06	0	ONLY WITH MANUAL TRANSMISSION POOR ROAD
500004	06	1	VERSION,CODE Z11 AND WITH PTO CODE N05/N07
500004	07	0	ONLY WITH AUTOMATIC TRANSMISSION POOR ROAD
500004	07	1	VERSION, CODE Z11

Table 5.5: Examples of stroke versions.

As I was told, one problem is that the six digit main numbers are getting exhausted. So the stroke versions are misused to extend the number space and to pack special versions together which have no relationship.

Stroke Version Interval

Stroke versions are divided up into 10 intervals, using some kind of fixed aggregation. There are ten stroke version intervals. Interval 1 contains stroke versions 1 to 10, interval 2 stroke versions 11 to 20, and so on.

The only place where the stroke version interval is *explicitly* used is for part records. A part record is attached to an interval and describes the ten stroke versions of the interval in parallel. See the section about special version part records below for details.

Special Version Images and Image Numbers

Special version images are used similar to the images used for model catalogues. See there for details. The tables used to store the image identifiers and image numbers are called SA_BILDTAFEL and SA_MAPS, respectively.

Part Numbers

The part numbers are used exactly the same way as described for model catalogues. See there for details.

Special Version Part Records (table SA_TPOS)

This is the main object type in the special version catalogues. A part record is attached to a stroke version interval using the primary key attribute INTERV. A part record describes if and in which amount a part number is used within the stroke versions contained in that interval. This association is similar to that between model part records and models. The string attribute ALLE_MENGEN contains ten 3 digit amounts for each stroke version. This corresponds to the association 'built into' shown in the diagram. Again, think of that as a matrix for each interval, where stroke versions are the columns and part records the rows.

As within model catalogues, a running number is used to distinguish between part records contained in the same interval. Two flags tell whether the part record is valid for vehicles with left or right hand drive.

Replacement of parts and interchangeable parts are handled the same way as for the part records for models.

A special version part record may also specify a part from a model catalog which is not built into a vehicle if it contains the respective special version. The sense behind this was, that one has to remove some parts to build in the special version component. This parts removal is the reason why the distinct documentation of models and special versions is also called the 'positive-negative' documentation method. The modern part authoring systems just model a vehicle as an abstract chassis into which alternate components and subcomponents can be build.

Special Version Footnotes

The footnotes for special version catalogues are organized the same way as those for regular model catalogues. See there for details.

5.2.6 Connections Between Models and Special Versions

Class diagram 5.4 shows the connections between model catalog objects and the stroke versions contained in special version catalogues.

There are two independently maintained connections: The table SA_VERWENDUNG (special version usage) lists all special versions usable with a construction group. This table is fed with data contained in model catalogues. In fact, the data is contained in a footnote having the magic footnote number 999. Another table, SA_UBM, lists all vehicle or aggregate models a stroke version can be

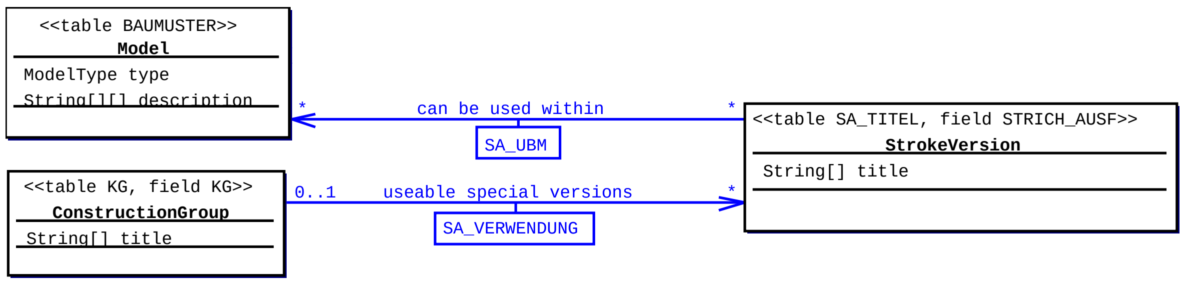

Figure 5.4: Connections between model catalogues and special version catalogues

built into. That table is fed with data from the special version catalogues input file. The set of models for a stroke version is coded into the attributes BR and UBM. The former identifies the model line and the latter is a string containing a list of models or model ranges '...BIS...', where 'BIS' is german for 'to'. Example: BR=730, UBM="409411BIS413414". The UBM field can sometimes be very long, e.g. more than 200 characters.

5.2.7 Supplementary Texts

The supplementary text class is shown on the class diagrams of both model and special version catalogues.

Its associated table BEITEXT contains a line of text in each language identified by a special attribute ADR_ERG_TEXT. This attribute sometimes begins with the construction group when the text is construction group specific. It also contains a julian date which tells when the record was last changed or when it was created. A supplementary text can be everything which makes sense for a part. It may for example be a more precise description of the part, a condition for its use, or a description where it's built-in. Some examples follow:

AUTOMATIC HEATING (BURLED WALNUT)
DIVIDED LENGTHWISE
TRITON GREY
USED WITH HITCHING MECHANISM
USED WITH "EXCLUSIVE" LEATHER TRIM, LEFT
USED WITH CENTRAL LOCKING MECHANISM, LEFT
UP TO THE END OF MODEL YEAR '91
ONLY APPLICABLE TO VEHICLES EQUIPPED WITH ONE OF THE FOLLOWING SA'S:
FROM PUMP HOUSING TO CARRIER ASSEMBLY
SCREENING PLATE TO CYLINDER HEAD COVER
MUST NOT BE EXCHANGED INDIVIDUALLY
FIRST EXHAUST STOCK OF OLD PARTS UP TO IDENT NO.

Supplementary texts are loaded into the database from a separate file. Very few supplementary texts are extracted from model or special version catalog input files. See next section for information about how data is loaded into StarDB.

5.2.8 Conversion of Legacy EPC Data to StarParts Database

This section describes how data is currently converted and loaded into the StarParts database. The conversion described here is based on several 'GNU Awk' scripts. Awk is a language designed especially for processing of textual data.

It is planned that the conversion process is switched over to using Java programs in the near future. Diagram 5.5 shows the input files, programs, and data flows of the conversion process.

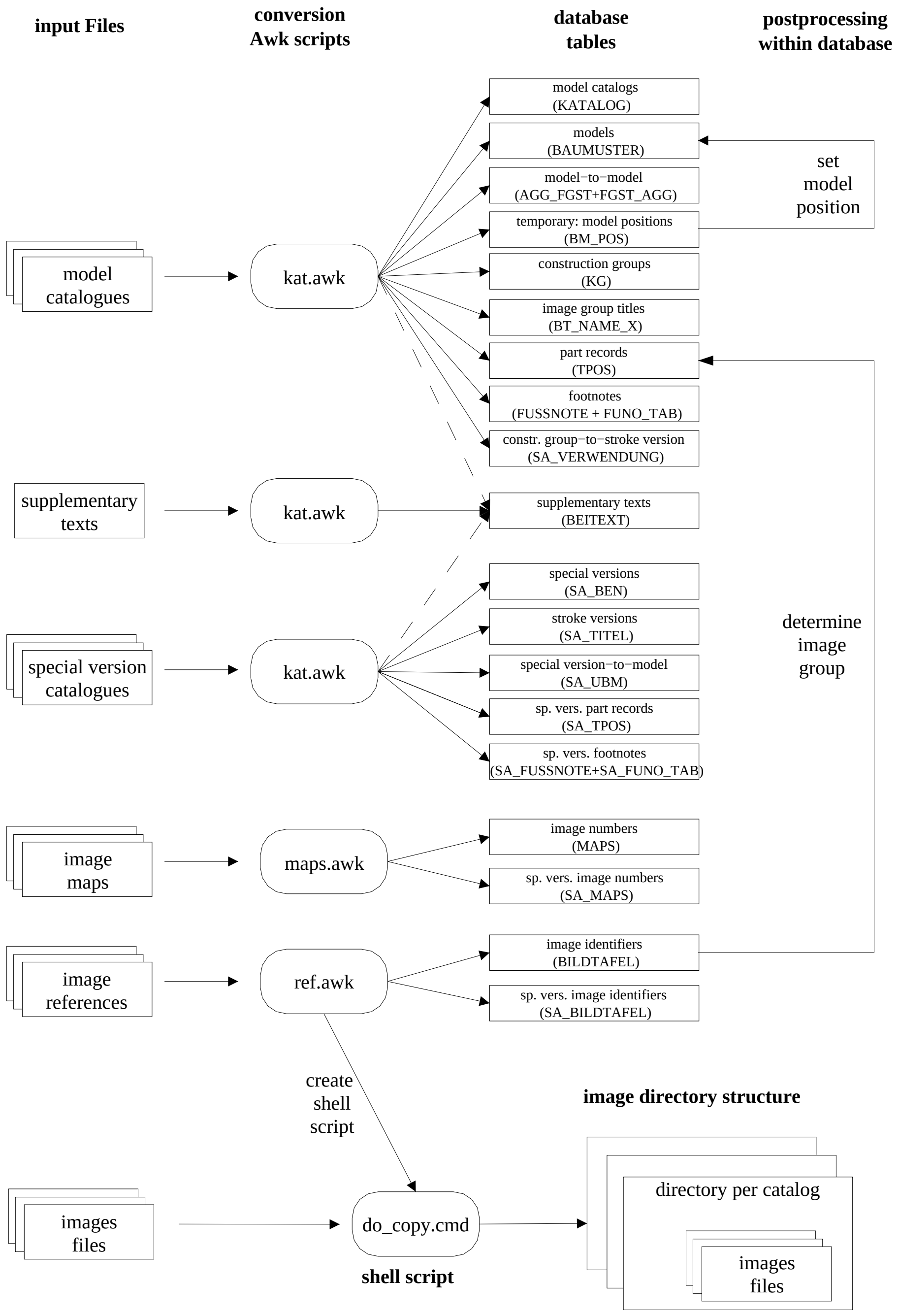
input Files
conversion Awk scripts
database tables
postprocessing within database
model catalogs (KATALOG)
models (BAUMUSTER)
model−to−model (AGG_FGST+FGST_AGG)
temporary: model positions (BM_POS)
construction groups (KG)
image group titles (BT_NAME_X)
part records (TPOS)
footnotes (FUSSNOTE + FUNO_TAB)
constr. group−to−stroke version (SA_VERWENDUNG)
supplementary texts (BEITEXT)
special versions (SA_BEN)
stroke versions (SA_TITEL)
special version−to−model (SA_UBM)
sp. vers. part records (SA_TPOS)
sp. vers. footnotes (SA_FUSSNOTE+SA_FUNO_TAB)
image numbers (MAPS)
sp. vers. image numbers (SA_MAPS)
image identifiers (BILDTAFEL)
sp. vers. image identifiers (SA_BILDTAFEL)
set model position
determine image group
model catalogues
kat.awk
supplementary texts
kat.awk
special version catalogues
kat.awk
image maps
maps.awk
image references
ref.awk
create shell script
images files
do_copy.cmd
shell script
image directory structure
directory per catalog
images files

Figure 5.5: Input files, programs, and data flows for conversion of EPC data to StarParts database

The reason why the Awk script ’kat.awk’ is drawn three times is that it’s invoked separately for each model and special version catalog file, and for the file containing supplementary texts. Placing it only one time might incorrectly suggest a coordination between input files. The arrow from the catalog conversion invocations to the supplementary text table is dotted because it is used very seldom.

Synchronization of input files is achieved simply by an assumption: it is assumed that input files which are available at the same time are mutually consistent. . .

Before data is fed into the online StarNetwork database, a temporary database is loaded to see if the input data was obviously incorrect or inconsistent or if something was broken during the conversion process. Note that the Awk scripts do not directly feed data into a database, but instead just generate importable files.

The data collected from all types of input files is always a full snapshot of some catalogues or of the supplementary texts. Almost always some primary keys have changed between snapshots. This is because most of the keys contain line numbers and other running numbers. So in order to fed the new data into the database, all records belonging to the changed catalogues will be deleted, and in the same transaction the new records are inserted into the database. The data volume of such a transaction is about 10-100 MB.

The next sections describe the conversion processes with more detail.

Model Catalog Conversion

Diagram 5.6 shows how a model catalog input file is structured, and which sections are fed into which table(s).

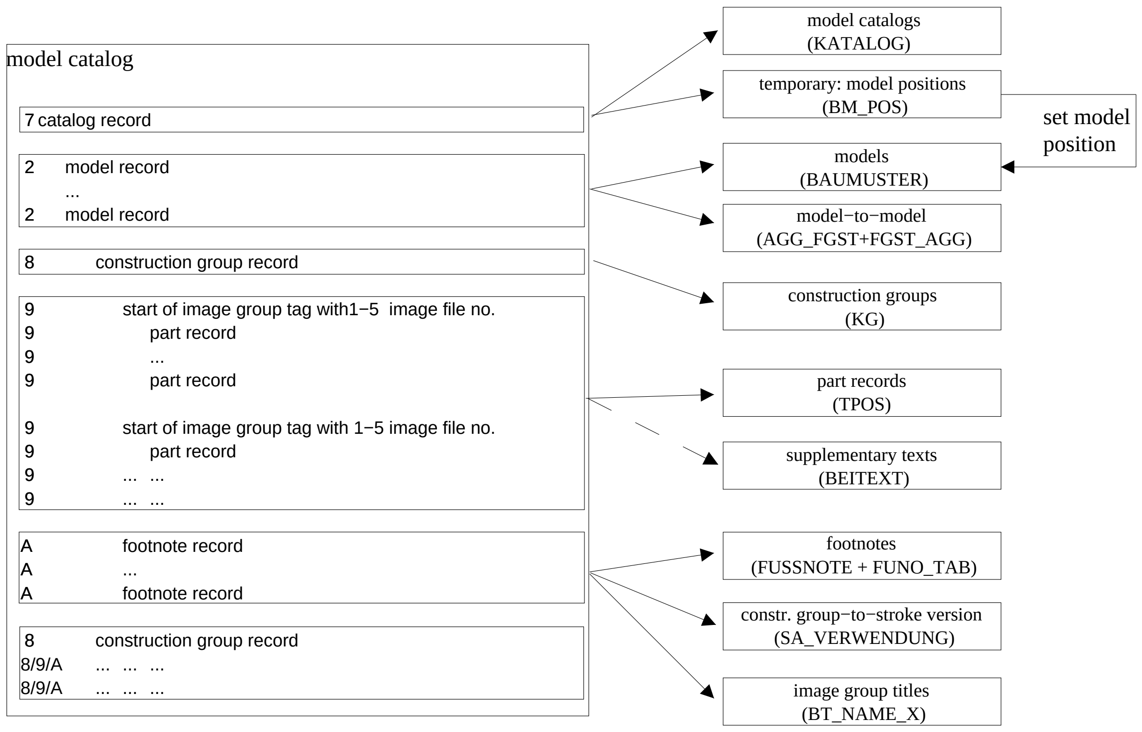
model catalog
7 catalog record
2 model record
...
2 model record
8 construction group record
9 start of image group tag with1−5 image file no.
9 part record
9 ...
9 part record
9 start of image group tag with 1−5 image file no.
9 part record
9 ... ...
9 ... ...
A footnote record
A ...
A footnote record
8 construction group record
8/9/A ... ...
8/9/A ... ...
model catalogs (KATALOG)
temporary: model positions (BM_POS)
set model position
models (BAUMUSTER)
model−to−model (AGG_FGST+FGST_AGG)
construction groups (KG)
part records (TPOS)
supplementary texts (BEITEXT)
footnotes (FUSSNOTE + FUNO_TAB)
constr. group−to−stroke version (SA_VERWENDUNG)
image group titles (BT_NAME_X)

Figure 5.6: Conversion of model catalog

The number on the left of each input record is the record type[1] as defined within the MB internal document “Interface ELDAS to EPC” [ELD97]. Every model catalog begins with a record type ’7’,

[1]The german abbreviation used for record type is SA=Satzart. Unfortunately, this is the same as those used for special versions SA=Sonderausstattung. This can be confusing when reading MB internal documents.

which contains all necessary information for the model catalog table (KATALOG). The information about the position for each model within part records[2] is also contained here, so an intermediate temporary table called BM_POS is used to store it (which is not really necessary). The data from this table is later transferred into the model table using SQL, as shown by the arrow 'set model position'.

After that initial record, several records type '2' follow. The data from these is loaded into the model table (BAUMUSTER). If the catalog is for chassis models, the chassis to aggregate mapping table FGST_AGG is filled. Else, data is supplied to the aggregate to chassis mapping table AGG_FGST.

The records following the model records are divided up into construction groups. Every construction group starts with a record type '8', which contains just all the data for the construction group table. What lucky chance …

The part records (type '9') within a construction group are further divided up into image groups. The start of an image group is defined by a record containing a special flag and 1-5 image file numbers. The first one is stored into the image file number field BT_NR[3] of all the following part records converted. The image group identifier field TU of the part records is later filled in using this image number combined with the data contained in the image references file.

Very few part records contain some additional text. In that case, a unique address is generated for that text with which it can be loaded into the supplementary texts table.

Within each construction group, footnote records (type 'A') are following after all the part records. Each footnote record can contain several lines of text. This is why the line numbering in the footnote database tables is split up among two columns: The first one, named FN_FOLGE, takes the running number contained in each footnote record, the second, named FN_ZEILE, counts the lines contained in one footnote record. If a footnote record has a special start marker, all footnotes following that are grouped together to form a table footnote until a footnote record with an end marker is reached.

Two special footnotes are evaluated: The footnote '1', which contains the titles of the image groups within a construction group, and the footnote '999', containing the special versions usable with that construction group.

Special Version Catalog Conversion

Similar to the model catalog conversion, diagram 5.7 shows how a special version catalog input file is structured, and which sections are fed into which table(s).

The structure of the file is similar to that of model catalog files. Each special version catalog starts with a record type '6', from which data is fed into the special versions table SA_BEN. After that, several records type 'C' describing stroke versions follow. These records can either contain model identifiers, in which case that information is fed into the special version to model mapping table SA_UBM, or title data, which is loaded into the stroke versions table SA_TITEL.

Supplementary text is extracted from special version part records as from model catalog part records, and footnotes are evaluated the same way as those for model catalogues.

Supplementary Texts

Most supplementary texts are loaded from a separate file. This file contains only type '5' records containing the supplementary text in all supported languages. These records contain also the date of last change or creation as a 5 character Julian date 'YYDDD' …

[2] See section describing part records !

[3] not to be mixed up with the image number attribute BILD_NR.

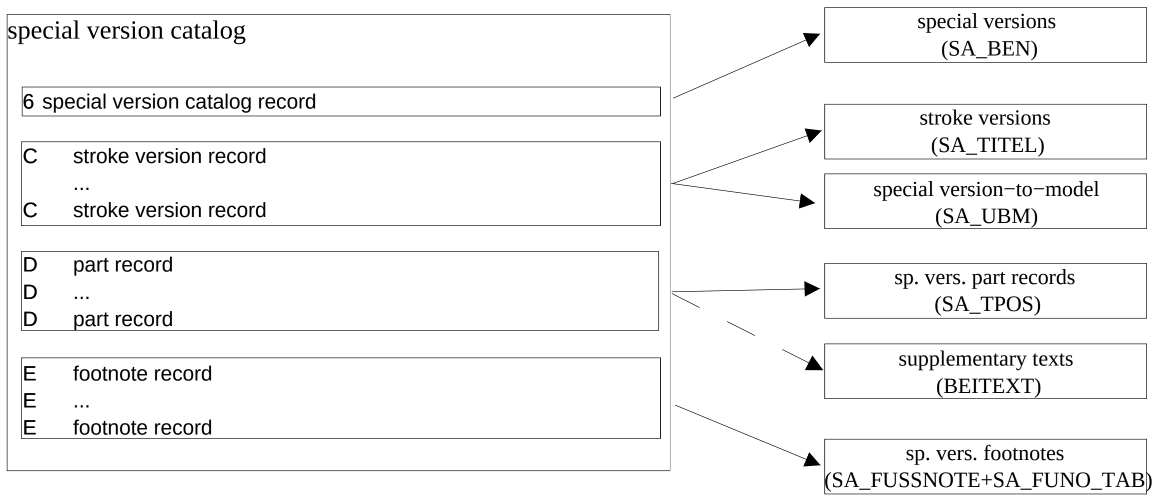
special version catalog
6 special version catalog record
C stroke version record
...
C stroke version record
D part record
D ...
D part record
E footnote record
E ...
E footnote record
special versions (SA_BEN)
stroke versions (SA_TITEL)
special version−to−model (SA_UBM)
sp. vers. part records (SA_TPOS)
supplementary texts (BEITEXT)
sp. vers. footnotes (SA_FUSSNOTE+SA_FUNO_TAB)

Figure 5.7: Conversion of special version catalog

Image Maps

Image map files are produced when the enclosing rectangles and image numbers of parts shown on images are detected with some kind of special OCR software (mentioned in 3.1). The image numbers are extracted from the image map files and stored into one of the tables MAPS or SA_MAPS.

Currently, the image maps are not used. They were generated directly by the OCR software, so the maps reflected the automatically detected image numbers found on the images, and not those that were corrected due to manual review.

Image References

Image references files contain a list of image files and image file numbers used by a catalog. It is used to fill the image reference table BILDTAFEL and to create a script which will copy the image files needed into the directory used for that catalog.

The image file number is also stored into the database, because it will later be used to fill the image group identifier field TU of the part records. The next section contains an example of these image reference files.

Image Files

are just copied to the appropriate location by the script created when the image references files were evaluated.

Note about Data Consistency

During use of the StarParts application, data from different tables is joined together using SQL queries. If data does not fit together, it normally will be omitted from the result sets of these queries. So if data is currently inconsistent or missing, some parts, images, special versions, footnotes, ... might not be accessible to the user, without any special notification about that fact.

5.2.9 Statistics for EPC Database

Currently, there are about 3000 model catalogues and about 12.000 special version catalogues available. About 1200 model catalogues are created for Europe, occupying about 6 GB, so the total size of the EPC database can be estimated to be about 15 GB. The total number of image files is about 100,000 (about 1 GB).

The reference installation I used covers 130 model catalogues and 773 special version catalogues. The total number of special versions found in the SA_BEN table is about 3580, but many contain only some note like 'see standard microfiche' in their title.

The size of this reference database is about 300 MB when stored within a MySQL database. Storing the data within an DB2 or Oracle database occupies between 2-3 times this space.

The following table show some statistics for the model catalogues. The minimum, average, and maximum counts are per model catalog.

entity	total	min	avg	max	size (MB)
catalogues	129	-	-	-	0.005
models	1215	1	9	40	0.6
construction groups	1952	1	15	46	0.2
image groups	7904	0	61	213	3
images	11438	1	89	322	0.5
image numbers	380963	8	2953	11970	19
part records	397149	26	3103	14514	93
footnotes	26989	1	209	1116	-
table footnotes	5805	0	46	392	-
table footnotes != 1	3871	0	30	125	-
footnote lines	175639	6	1362	10388	14
table fn. lines	746381	0	5786	26749	110
table fn. lines != 1	78762	0	610	10973	9

Table 5.6: Sizes of model catalog tables.

The 'table footnotes != 1' will be explained in the next section. One should know that aggregate model catalogues are much smaller than those for chassis models. Aggregate model catalogues contain 1-2 construction groups for axis or transmission models, and about 10-15 for engine models. Chassis model catalogues contain between 20 and 50 construction groups.

Table 5.7 shows statistics for the special version catalogues. I omitted the minimum and average values for the special version catalogues as they are not very meaningful. This is because many special version catalogues do not contain all types of entities.

The tables show that the special version catalogues are much smaller than the model catalogues. Table 5.8 shows the sizes of the mapping tables between these two catalog types and between the different types of models. The mappings between chassis and aggregate models are very small, but those between models and special versions are quite large.

The image files for the catalogues in the reference installation occupy about 110 MB when stored in one directory per catalog, and only 65 MB where stored in one directory (redundant images removed). One image is 10 KB on average, the smallest image in the reference installation was 1.7 KB, the biggest 36 KB.

entity	total	max	size (MB)
special versions	3580	-	1.6
stroke versions	17394	93	5.0
images	2382	75	0.1
image numbers	22469	719	0.8
part records	47082	1731	9
footnotes	5096	145	-
table footnotes	365	26	-
footnote lines	117159	3912	10
table fn. lines	2422	283	0.3

Table 5.7: Sizes of special version catalog tables.

entity	records	size in MB
chassis -> aggregate	4223	0.2
aggregate -> chassis	7339	0.7
constr. group -> stroke version	187370	8
stroke version -> model	40729	3

Table 5.8: Sizes of cross reference tables.

Last but not least, the supplementary text table is about 26 MB, and contains about 120,000 records. But these counts do not increase when more catalogues are installed.

5.2.10 Improving the Data Structures and Conversion Processes

Improving Consistency Between Catalogues and Images

The image file numbers (BT_NR) are used after the conversion of a model catalog to determine the image group a part record belongs to. The following table 5.9 gives an example of such a mapping, contained in the mentioned image references files.

Image file number (BT_NR)	01	02	03	04	05	06	07	08
Image group (TU)	517	530	545	545	545	560	580	590

Table 5.9: Example mapping contained in image references file.

The next table 5.10 shows the start of image group tags used within the catalog, and the image group (TU) the part records found after that tag will be assigned.

As the conversion is currently done, the first image number of each image group tag is stored into each of the following part records. After the reference file is loaded into the image reference table BILDTAFEL, the image group (TU) of each part record is initialized with the TU found in this table.

Now, as already mentioned, there is a synchronization gap between the images coming together with the reference files from the RGBG system and the parts catalogues coming from either ELDAS,

original tag	assigned TU
01	517
02	530
030405	545
06	560
07	580
08	590

Table 5.10: Example of start of image group tags mapped to image groups (TUs).

MAD, or Bell&Howell. Sometimes, image groups (TUs) in the catalog are rearranged. We consider as an example here that ’560’ was deleted, and ’595’ was newly appended. Note that rearrangements in such ways do actually happen. The following table 5.11 shows, how TUs are wrongly assigned when the old reference table is used. This has no immediate noticeable effect, as e.g. an unassigned image group (TU) would be.

original tag	correct TU	assigned TU
01	517	517
02	530	530
030405	545	545
06	580	560
07	590	580
08	595	590

Table 5.11: Example of wrong assignment of image groups (TUs) to start of image group tags.

Now, (nearly) every construction group contains a table footnote number one. This footnote contains a list of all image groups along with their names found in that construction group. As observed, these image group identifiers (TUs) are stable among such rearrangements. The footnotes are updated correctly. Extracting the image group identifiers (TUs) from that footnotes will increase consistency, as version errors like that described above are now detectable.

But, I was told that there are two arguments against this. The first one was, that this feature is undocumented. Second, there are some, but very few, construction groups without a footnote number one.

I don’t think that these arguments are justified, because of the following reasons:

- There must be some reason that this footnote is contained in nearly all construction groups, and that these footnotes are carefully maintained. Since this feature is apparently used by someone (maybe Bell&Howell ?), someone has to complete the documentation !
- The awk script implicitly extracts the image group names (stored in the BT_NAME_X tables) from these footnotes. Since StarParts includes these in a join operation, using the current conversion process the images from construction groups without the footnote #1 will be omitted *silently!*

- Those few construction groups containing no footnote #1 can be repaired even manually. But it would be better to determine why those construction groups lack this footnote, and, where necessary, fix it in the standard authoring process.

Detecting versioning errors if only the images itself are modified is difficult. But inconsistencies produced by these are not as worse as using the wrong images, like shown in the example above.

Reducing Database Size by Removing Unreferenced Footnotes

This again affects the footnote number one. All information contained in those footnotes is extracted to the image name tables, which store the information in a more compact form than the footnotes. The footnotes contain many empty lines which are also stored within the database.

The footnotes #1 are seldom referenced. Of 2,000 construction groups, a footnote #1 was referenced only within 5. The question is, whether these references are actually correct, since it makes few sense to reference these footnotes from a part record.

Removing these unreferenced footnotes from the FUNO_TAB table in our reference installation of about 130 catalogues saved over 100 MB space - 92% of the space occupied by that table and 34% of the size occupied by the *whole* database !

I couldn't figure out any reason why this should not scale up to about 2.2 GB saved space for all 3,000 parts catalogues. This wasted space might not hurt much for the main database and MLC databases, but it does for a local installation at a dealer's or workshop's site.

Avoiding the Renaming of Table Footnotes Between Releases

In the original version of the conversion script, the grouping number (BLOCK_ZAEHLER) used for table footnotes was just one running number for the whole catalog. This was bad for change detection and the generated updates, since the insertion or deletion of table footnotes affects all records following that modification.

So I modified the awk script (just 2 lines) to use the first footnote number in a table footnote as grouping number. This always works and produces stable grouping numbers among releases.

Reducing Space Occupied by Image Files

The image files are currently stored in one directory for each catalog. This produces about 40% overhead since many images are shared among several catalogues.

5.2.11 Changes Between Releases

This section illustrates some of the more complex changes between releases. These changes were challenging for the analyzer tool, as these changes are the reason why heuristics had to be introduced to detect changes.

Simple Example for Update of Part Records

Table 5.12 shows a quite amusing example of part records that were obviously renumbered just for fun. Between the unchanged records 4000 and 4800, the part records were assigned new running numbers, and the code condition field of one of the records was updated.

nr.	part number	description	code	FN1	FN2	all quantities (ALLE_MENGEN)
4000	A 1703210004	FRONT SPRING		2	-1	002002002002002002002002002nul. . . nul
4500	A 2103211204	FRONT SPRING		2	4	002002002002002002002002002nul. . . nul
4550	A 2023211804	FRONT SPRING		2	4	002002002002002002002002002nul. . . nul
4600	A 2103211404	FRONT SPRING		2	-1	002002002002002002002002002nul. . . nul
4800	A 2023211504	FRONT SPRING	482	2	-1	002002002002002002002002002nul. . . nul

Was changed to the following in the new release:

nr.	part number	description	code	FN1	FN2	all quantities (ALLE_MENGEN)
4000	A 1703210004	FRONT SPRING		2	-1	002002002002002002002002002nul. . . nul
4100	A 2103211204	FRONT SPRING		2	4	002002002002002002002002002nul. . . nul
4200	A 2023211804	FRONT SPRING	-488	2	4	002002002002002002002002002nul. . . nul
4300	A 2103211404	FRONT SPRING		2	-1	002002002002002002002002002nul. . . nul
4800	A 2023211504	FRONT SPRING	482	2	-1	002002002002002002002002002nul. . . nul

Table 5.12: Example showing running numbers of part records changing between releases.

The part records shown are not complete; a full part record contains 45 attributes. I will explain some of the fields to give an idea here of what part records are: The attributes FN1 and FN2 are two from the six reference fields to footnotes. The referenced footnote number 2 is a table footnote listing points for special versions which one must add on to ascertain the front springs. The footnote is too big to be shown here, but it e.g. says that you have to add on 8 points for an air conditioner and 3 points for an automatic transmission. The footnote number 4 says just ’Vehicles, Australia’. As you can see in the all amounts field, the front springs are build two times into all the 9 vehicle models this catalog is valid for, which is not very surprising. The newly introduced condition code expresses the fact that this front spring is not built into vehicles having the 488 distribution code which means ’sporty tuning’. This code can be found for the respective sporty tuned cars in the vehicle data cards from the FDOK database.

Reorganization of Images

As already mentioned, image groups and images are sometimes reorganized. The following table 5.13 shows the image file numbers (BT_NR) and image group identifiers (TU) contained in construction group 82 of three releases of the model catalog ’45H’.

The image file number (BT_NR) 99 is apparently used if some images are not yet available. The corresponding image number (BILD_NR) fields were set to ’—–’ to express the fact that the part records are not shown on any image.

Between the first two releases, 330 part records of about 1000 in that construction group were updated. Due to the reorganization, the running numbers of 220 part records were changed, as they were moved to new locations.

Adding New Vehicle Models

As it can get only worse, between the second and the third release of the last example, 5532 part records of all 8632 contained in the whole catalog were updated. This happened because a new model

February	’98	March	’98	May	’98
BT_NR	TU	BT_NR	TU	BT_NR	TU
…	…	…	…	…	…
09	285	09	285	09	285
101112	315	101112	305	10	305
		99	320	1112	320
				13	327
13	332	13	332	14	332
1415	335			15	335
		1415	336		
				16171819	338
16	345	16	345	20	345
		99	347	21	347
		99	360	22	360
17	445	17	445	23	445
18	465	18	465	24	465
19	510	19	510	25	510
20212223	530	20212223	530	26272829	530
24252627	575				
28	605	28	605	30	605
29	610	29	610	31	610

Table 5.13: Reorganization of image groups (TUs) between releases

was introduced within that catalog. Of course, all ALLE_MENGEN strings had to be updated to reflect the fact that the parts are build into that new model.

It happens also, that the assignment of models to catalogues changes. When a catalog is full, it will be split, and the respective models that are most similar are put together in one of the new catalogues.

This must also be considered when designing the subsets for the EPC database, because the assignment of models to catalogues is not fixed. If vehicle models would be used for subdivision, such a reassignment would also affect the subdivision rules of the database, which introduces an additional complexity. An approach of subdividing the EPC database will be described in section 5.2.15.

Updating Table Footnotes

Table footnotes can have up to 100 lines for each of the 6 languages. So if a line is inserted at the beginning of a table footnote, all 600 lines following that may have to be updated.

5.2.12 Updates to Images

I already mentioned that the images are not stored in the database, but in the filesystem. An image maybe shared among several parts catalogues. Whenever an image is revised, a new version with a different filename is created. The image file format compresses very well; in fact, trying to compress an image with ’gzip’ or ’bzip2’ may make it slightly larger. For that reason and because the binary data of the new version of an image has nothing in common with that of the old version, I decided not to implement a change detection for images.

I also do not generate Smart Updates for images. After the database updates to the image references table have been applied, the target node can download the images that are newly referenced from the source node. In an enhanced approach, the source node would send the necessary images to the target node in advance.

There are other approaches for selecting a strategy of distribution and local caching of images as well: For nodes that have a persistent online connection, it makes sense to load the images on demand. Another possible approach might be to store only those images locally that belong to the most used construction groups. It will be no big problem when an image is neither locally nor remotely available due to a network or remote server outage, since the StarParts application can do without the missing image.

5.2.13 Using Timestamp Replication

As shown in the previous chapter, many changes are done in a 'vertical manner', due to the running numbers used for part records, footnote lines, and image file numbers. If one would assign timestamps only to records, this would lead to an exceeding number of changed records. But some steps could be undertaken which would soften this.

A unique identity is needed when timestamp replication should be effective. Else the moving of records would result either in two delete, insert, or update operations. The following steps describe how the EPC database could be redesigned to allow a not-to-ineffective timestamp replication.

- Part records could be redesigned to get a unique identity. The analyzer tool can handle this, as it is able to match the new part records and footnotes coming without identity to the old records in the reference table which then contain an identity attribute.

- The running number (LFD_NR) and image file number (BT_NR) fields of part records should not be replicated, as they are not really needed. The StarParts application can sort the part records by the image number (BILD_NR) field, which results in a very similar order as produced by using the running number (LFD_NR).

- Two timestamps should be used for part records, one for the 'all quantities' (ALLE_MENGEN) field, and one for the rest.

- Footnote lines are also redesigned to get a unique identity. Two separate timestamps, one for all the line numbers and one for the text field, should be used. If lines are inserted, only the changed line numbers would have to be transmitted. The line numbers can't be omitted, since then the order of lines could not be determined by the StarParts application.

- Records must not be deleted immediately from the database, as then the information about the deletion is lost. Instead, they would have to be flagged as deleted, and the timestamp must be updated.

- Some database triggers would have to be introduced (only within the StarDB) to handle the updates to records in tables with two (or more) timestamps, since the database normally updates only one timestamp automatically.

By redesigning the data structures this way, a bug in the current implementation of StarNetwork would also be fixed. StarNetwork uses the running number of part records in its shopping list which is saved between sessions. If the running number of a part record is changed between two sessions, the shopping list contains inconsistent information which may lead to unpredictable results.

The next section contains some update sizes which allow estimating the transmitted amounts using this replication technique.

5.2.14 Sizes of Updates to EPC

This section compares several techniques of representing updates by the size of the changes that are produced. The information found here was taken from the history of catalog and image updates that was maintained by debis since March 1998.

Model Catalog Updates

Within the 6 month between March and September 1998, about 1,950 model catalogues were updated. The total compressed size of these catalogues was about 230 MB. I selected a smaller subset containing different types of catalogues to test the analyzer tool and estimate the total size of updates for all catalogues. Table 5.14 shows the selected catalogues, how often they were updated within the 6 month, and the total size of updates using different methods for representing updates.

catalog	a vehicle/aggregate model	#updates	snapshot	record based	Smart Updates
01C	G 210-16, Transmission	4	336 KB	197.0 KB	106.6 KB
04S	ML 230, All Activity Vehicle	6	1121 KB	270.8 KB	129.4 KB
07B	GL 68/20 A-5, Transmission	4	156 KB	8.6 KB	6.8 KB
07C	GL 76/27 A-5, Transmission	4	127 KB	6.4 KB	5.7 KB
09S	W 4 A 020, Automatic Trans.	4	185 KB	8.2 KB	7.6 KB
19T	OM 606, Diesel Engine	6	708 KB	33.8 KB	22.9 KB
19X	M 112 E24, Gasoline Engine	4	239 KB	108.1 KB	49.9 KB
30L	1831,1835, Truck	4	1461 KB	311.7 KB	103.5 KB
35A	Sprinter 208 D	7	1860 KB	194.1 KB	96.3 KB
44V	C 180, Sedan	8	3678 KB	466.0 KB	167.0 KB
45H	C 180, Station Wagon	8	2353 KB	411.9 KB	154.3 KB
45P	E 200 Diesel, Sedan	8	3314 KB	486.6 KB	188.9 KB
45Y	CLK 200, Coupe	6	2191 KB	362.0 KB	186.1 KB
502	OM 602, Bastic Engine	4	830 KB	33.3 KB	22.2 KB

Table 5.14: Compressed sizes of updates to selected model catalogues using different methods.

All sizes in the tables are for compressed data respectively updates; the sizes for uncompressed data or updates are 20-25 times larger.

The ’snapshot’ column tells how much data one would need to transfer if the whole catalog would be transmitted whenever it was changed.

The ’record based’ column tells how big the updates are if whole records are transmitted whenever they are changed. This includes records marked as changed due to relocation of records (changes in the primary key fields). This size is an estimate for the size of updates transmitted when using timestamp replication.

The 'Smart Updates' column contains the sizes of the updates that are currently generated by the analyzer tool.

Based on the sizes in table 5.14, the total amount of updates for the six month and the average amount of updates per week was estimated (table 5.15).

	snapshots	record based	Smart Updates
6 month	230 MB	40 MB	15 MB
per week	9 MB	2 MB	0.6 MB

Table 5.15: Estimated compressed sizes of updates to all model catalogues using different methods.

These estimated sizes are quite small, but it has to be considered that revised catalogues are currently released only every few months[4]. This might be sufficient for the CD distribution, but it will probably have to be changed for online access with StarNetwork or for distribution via PAID, as else the catalogues are already outdated when the updates are generated.

When comparing the records update approach with the Smart Update approach, the planned conversion from the new authoring systems back to the EPC format has to be considered. Relocation of records will then happen more often than currently, so the updates that are record based will become much bigger.

Special Version Catalog Updates

The total amount of special version catalog snapshots in the six month mentioned above was about 70 MB. The size of the generated Smart Updates for those 70 MB is only about 5 MB. Since special versions are not used for new vehicles anymore, I don't expect this size to increase.

Images

Images that were revised or newly created are delivered in archive files together with the mentioned image reference files. Table 5.16 shows the number and total sizes of changed images between May and September 1998 per archive file.

The 'for days' column tells for how many days changed and new images were gathered for an archive file. The average amount of new and updated images is 3.3 MB per week.

5.2.15 Subdividing the Database for PAID

Since PAID does only install some subsets of the data locally, a good way of subdividing data has to be found. The requirements for subdivision are found in section 4.4.5 describing the Smart Updates.

It is important to mention here that there are two different views to the subdividing of data, namely the user's view and the system's view. The user must not be confronted with any details about how data is divided up into subsets. This section is split up into a part describing the system's view, one for the user's view, and one describing how to generate the former from the latter.

[4]The catalogues in table 5.14 were selected because they were released more often.

archive date	for days	number	total size
1998-05-08	33	1707	16.7 MB
1998-05-23	15	461	4.5 MB
1998-05-30	7	420	4.1 MB
1998-06-06	7	460	4.5 MB
1998-06-13	7	135	1.3 MB
1998-06-20	7	374	3.7 MB
1998-06-27	7	267	2.6 MB
1998-07-04	7	330	3.2 MB
1998-07-11	7	482	4.7 MB
1998-07-18	7	268	2.6 MB
1998-07-25	7	679	6.6 MB
1998-08-01	7	811	7.9 MB
1998-08-08	7	125	1.2 MB
1998-08-22	14	234	2.3 MB
1998-08-29	7	154	1.5 MB
total	146	6907	67.5

Table 5.16: Number and total sizes of new and changed images.

System's View on Subdividing EPC

Class diagram 5.8 shows how the EPC database is divided up into subsets.

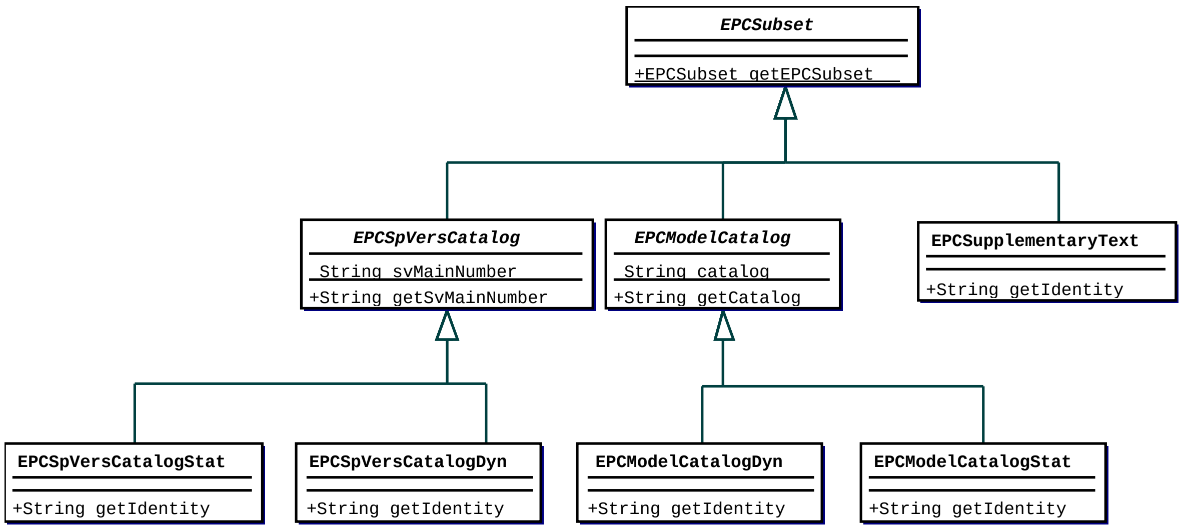

Figure 5.8: Subsets of EPC database

Each model catalog and special version catalog consists of two subsets: a 'static' and a 'dynamic' one. This is because a node probably needs nearly all static subsets, but fewer of the dynamic. This is because the former are needed for the manual vehicle identification and for generating the system's view from the user's view.

The static subset of a model catalog contains the data from the tables KATALOG, MODEL, FGST_AGG, AGG_FGST, and SA_VERWENDUNG. As already described, these tables contain the information about the area, vehicle classes, and models a catalog is used for, the mapping between chassis and aggregate models, and information about which special versions can be used for each construction group.

The static subset of a special version catalog contains the tables SA_BEN, SA_TITEL, and SA_UBM, containing information about the area and vehicle classes a special version is used for, the possible stroke versions of a special version, and the models these can be used for. These tables are solely used to compute the system's view from the user's view.

The tables of the static subsets are all relatively small, so it won't hurt to install most of this data locally on each node. The dynamic subsets of the catalogues cover all the other tables, and depend on the static subsets. The EPCSupplementaryText subset refers to the table BEITEXT, and is needed as soon as at least one dynamic subset is installed, since then there are part records referring to that table.

Class diagram 5.9 shows, how the local data description for selecting and customizing the Smart Updates is represented.

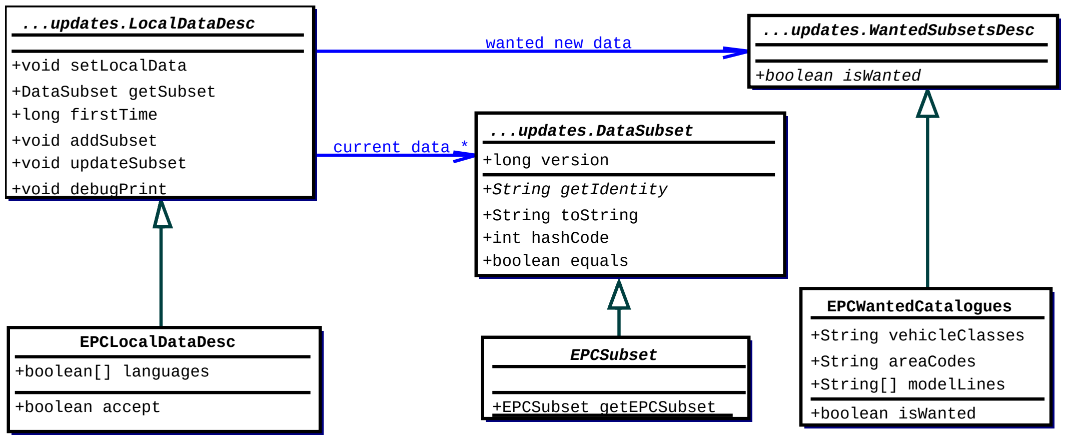

Figure 5.9: Local data description of EPC database

The EPCSubset is a subclass of the general DataSubset class. The local data description contains an additional field 'languages', which tells which languages a node wants to use. Using only one language instead of all six currently supported by EPC saves about 33% space[5]. The customized updates are smaller for only one language as well.

If new catalogues are introduced, the EPCWantedCatalogues object associated with the local data description LocalDataDesc tells whether a local node wants to get the updates installing the new catalogues. Only the areaCodes and vehicleClasses attributes are evaluated for the static subsets. Additionally, the modelLines attribute specify which dynamic subsets are requested.

The division into static and dynamic subsets also enables a notification of new model lines that are introduced, since the static subsets of the corresponding catalogues are installed automatically if they fit into the selected area and vehicle classes.

It is important that the versions of the static and dynamic subsets match, since e.g. the position of a model within in the part records may have changed between releases. It is therefore important not to mix remote and local access of static and dynamic subsets of the same catalog. The StarNetwork ap-

[5]Without the removal of the 'footnotes #1' mentioned earlier, this saves about 48% space.

plication may access the static subsets locally for manual vehicle identification, but for parts retrieval, it may only access the static subset locally if the dynamic subset is installed as well.

Maybe the idea of a static and a dynamic subset should be dropped in favor of just installing the appropriate data needed for manual identification and to determine the needed catalogues redundantly. But note that this is equivalent to not allowing the access to the static subsets for parts retrieval.

User's View on Subdividing EPC

The user simply selects the languages and vehicle classes (e.g. only passenger cars) he wishes to install. The appropriate countries/areas should be determined automatically, but it should be changeable by the user. Additionally, the user should be able to choose to install only a subset of model lines locally.

Generating the System's View from the User's View.

These are performed in several steps:

1. The needed static subsets of model and special version catalogues can be determined directly from the given area and vehicle classes, since the corresponding tables KATALOG and SA_BEN contain this information. They have to be installed before the next steps can be performed.

2. The dynamic subsets are determined by first selecting all model catalogues for the given model lines. After that, the chassis to aggregate mappings are used to determine the aggregate catalogues which are used for the selected model lines.

3. The construction group to stroke version table is used to determine all special versions and stroke versions which can be used with the determined dynamic model catalogues.

4. If at least one dynamic subset is installed, the supplementary texts have to be installed.

This process works the same way if it should be possible to select individual models instead of only whole model lines.

Calculating the locally installed subsets in that way allows straight redirection of the queries Star-Network currently executes on the data.

5.3 FDOK - StarIdent

This section describes the vehicle data card database StarIdent, which was derived from the legacy FDOK database, which is a IMS/DB hierarchical database.

It has a very simple database schema. One big table holds all the vehicle data cards; three very small naming tables map the numeric codes used within the vehicle data cards to human readable descriptions.

5.3.1 Utilization and Contents of Vehicle Data Cards

The vehicle data cards are used mainly for parts identification. One needs to know what distribution codes (e.g. 580 - air conditioning) and special versions (see EPC) are valid for a specific vehicle, and which engine, transmission, axis, etc. models were built into that vehicle. That information can then be used to preselect the appropriate parts from the parts catalogues.

The following are the particular attributes found in a vehicle data card:

WHC The international manufacturer code. Mostly 'WDB' (85%), but there are others as well, e.g. VSA, 4JG, WDC, NMB.

FIN_RUMPF This is the main vehicle identification number. This uniquely identifies a vehicle and therefore also a vehicle data card. The complete FIN would start with the manufacturer code, but that had been put into a separate field. The first 6 digits of this attribute are the model line and the model. The last places are just a running number.

VIN This is a second vehicle identification number. It is used in some countries, e.g. the United States. It is redundant with the FIN and can be calculated from that.

MOTOR The engine identification number. The first 3 digits are the model line of the engine.

GETRIEBE The transmission identification number. Starts with model line as well.

AUFTRAG This is the order number of the vehicle. The digits 3-5 are a country / branch code describing where the order for the vehicle was placed.

VERSORG_DAT This is a timestamp telling when the data card was received from the original FDOK database.

OBJECT This binary field contains a compressed, serialized Java object.

The attributes above are stored redundantly outside the object in the database table. Appendix B.1 contains the full StarIdent schema. The following are some of the attributes found within the Java object:

CODES Contains the distribution codes. The numeric codes found here are either looked up in the table 'fdk_code_ben_pkw' for passenger cars or in the table 'fdk_code_ben_nfz' for utility vehicles. For passenger cars, these codes consist of 3 digits, and are only valid for a certain time interval within which the vehicle model was produced. When looking up the description, the production date of the vehicle is needed. For utility vehicles, alphanumeric codes are used, which cover 3 places as well. When looking up the description of this codes, the sort (SPARTE) of the utility vehicle is needed to select the appropriate code description.

SAs This are the special versions used for a vehicle. This is a list of construction groups (see EPC) with the respective special versions used. A special version may be used more than once. For utility vehicles, this list can contain more than 1,000 different special versions with their respective usage count.

Additional identifiers found within the gzipped Java object are color codes, the name of the vehicle model, the manufacturers of the lamps and tires, to name but a few. For utility vehicles, there are even more items, including the axis models, additional transmissions, the steering model, etc.

5.3.2 Conversion from FDOK to StarIdent Database

This is also very simple. The new and changed vehicle data cards are coming from FDOK in a file format which I call, in absence of an official name, the 1,2,3 format.

There are three types of records (lines) for each vehicle data card contained in these files. Records marked with a ’1’ at the beginning contain all attributes of a vehicle data card within fixed fields, except the special versions and distribution codes. Zero or more type ’2’ records following that contain the special versions and the respective construction groups they are used for. Following that, a type ’3’ record lists all distribution codes for the vehicle. The vehicle identification number (FIN) is contained in all records after the record type code.

The files are read by a small Java utility which evaluates the records, builds the vehicle data card object (FdkDataImpl), and writes it to the database. It first tries to execute an SQL INSERT for each card, and when that fails (which means it was a changed card, not a new), it executes an UPDATE.

5.3.3 Size of FDOK, and How It Can Be Reduced.

Currently, the StarIdent database contains vehicle data cards for about 11,000,000 passenger cars and 4,000,000 utility vehicles. The current size was estimated to be about 21 Gigabytes.

Most attributes of the vehicle data card are stored in the compressed serialized Java object. In the format currently used, the average size of this object is about 840 bytes for passenger cars and 2,660 bytes for utility vehicles. But I found a way to reduce the size of FDOK by nearly 9 GB by altering this format in a backward-compatible way, which means that the objects stored already in the database can still be read.

StarNetwork uses the standard Java Serializable interface to serialize the Java objects. But that produces a lot of overhead when serializing Java objects, because it writes the names and types of all attributes found in an object to the serialization stream. This is done to enable handling different versions of a class automatically. Normally, attribute and type names are written only once for each class for which instances are serialized. But in our case, where for each vehicle data card a new serialization stream has to be created, this is done for *every object*, which produces a lot of overhead. After compression, the overhead caused by this mechanism is nearly 600 bytes per object.

There is an alternative to using the Serialization API, which is called the Externalization API [JOS98]. Using the latter, it is the programmer’s responsibility to determine the format and versioning of the serialized objects. Externalization can be used as a drop-in replacement for serialization, there is no need to change the rest of the application. In fact, I didn’t even had the possibility to do that.

In absence of the StarNetwork sourcecode, I subclassed the vehicle data card (FdkDataImpl) to a new class called FdkDataImplExternal which implements the Externalizable interface. This also allows to still use the ’old’ data cards stored in the database, as the deserializing stream determines the class of the serialized objects and uses the right deserialization mechanism. The system doesn’t even know that there is a subclass when doing serialization operations with the superclass. I then created a little tool which used the Reflection API [JRef98] to generate most of the necessary code for externalization automatically. Given the number of attributes stored in the vehicle data card (about 50), this was the safest way to avoid introducing bugs when switching to externalization. StarNetwork works fine with the new data cards.

The following table shows the new sizes of the vehicle data card objects, and the total saved space for the whole StarIdent (FDOK) database.

vehicle type	old size (bytes)	new size (bytes)	saved (relative)	saved (total)
passenger	840	333	61%	5.2 GB
utility	2660	1863	30%	3.0 GB

Table 5.17: Size reduction of vehicle data cards

For branches which only use and store vehicle cards for passenger cars, the needed disk space is only a little more than one third of that needed before.

Note that this is still far from optimal. Given the number of vehicle data cards stored in the database, 65 bytes overhead per vehicle data card will result in 1 GB space for all vehicles. The long package path of the vehicle data card Java class (FdkDataImpl) occupies about 50 bytes (*after* compression), and so it takes about 0.8 GB to store this pathname redundantly for all vehicles.

For the dealer's sites, I suggest using a database system which is able to compress the data cards itself. As will be shown in the next section, multiple vehicle data cards can be compressed much better than a single. Storing all vehicle data cards into compressed flat files would result in e.g. one 2.6 GB file holding all 4,000,000 utility vehicle data cards and a 750 MB file containing all 11,000,000 passenger vehicle data cards. This is small enough to enable PAID to use a database system like e.g. TransbaseCD [Tra98], which is able to *transparently* handle the old data stored e.g. on a DVD and the new data coming as Smart Updates from the network. Important for such an approach is that it must be scalable, else we will run into the same problems as the CD distribution has today. TransbaseCD for example is able to handle several DVDs or CDs, so adding a second DVD drive (or something else) is no problem. But perhaps the network connections will be good enough for online access when that would be needed.

If the approach of using a compressing database is not possible or not feasible, the StarNetwork application and the load tool should be at least adopted to write the vehicle data cards directly to the compressing stream opened onto the database, instead of calling the write/readObject methods on an object output stream wrapped around that. The methods provided for externalization can be used for that purpose without any modification.

Also, the attributes that are stored already outside the object (e.g. the FIN) in the database do not need to be serialized, as they are when using the current method. The best solution would be if the vehicle data card class is given some methods to read/write itself in the appropriate way to the database.

I estimate that these two steps will save yet another 2 GB, perhaps even 3. One must also take into account that the database will grow slower when the vehicle data cards are smaller.

5.3.4 Sizes of FDOK Updates

Incremental updates are currently arriving every few weeks from the source FDOK IMS database. The following table shows the size of the updates generated by the analyzer tool. The table is organized into vehicle type and area; the counts and sizes are for a period of 5 weeks. An update originating at the legacy FDOK database always overwrites the contents of the vehicle data card, so there is no difference between new and changed vehicle data cards coming from there.

vehicle type	area	SU-size	#vehicles	DB-size	compressed
utility	Europe w/o	9.8 MB	23,000	43.0 MB	40 MB
	Germany	9.4 MB	15,000	34.0 MB	31 MB
	others	0.8 MB	2,000	2.1 MB	1.9 MB
passenger	Europe w/o	2.2 MB	35,000	14.5 MB	9 MB
	Germany	3.8 MB	48,000	20.0 MB	14 MB
	America	1.2 MB	21,000	10.0 MB	6 MB
	others	0.2 MB	2,000	0.9 MB	0.6 MB

Table 5.18: Sizes of FDOK updates

’SU-size’ is the the size of the generated Smart Updates. ’DB-size’ is the net size occupied by the vehicle data cards in the database, without overhead introduced by the DBMS. The ’compressed’ column tells what happens when the vehicle cards are compressed in the format they are stored in the database. If one would use a commercially available database replication technique, one would have to transmit at least this size, given that the replication technique supports compression at all.

Also, as can be calculated from these amounts when considering that a few data cards are not new, the StarIdent database grows by about 1 GB every year. Note that this size and all the sizes shown on the table include my switch from the Serialization API to the Externalization API introduced in the previous section. When using the standard serialization, it will grow by more than 2 GB every year.

The reason why the ’compressed’ size is bigger than the Smart Update size is that the vehicle data cards are already compressed. I tried several compression techniques[6], but the result was always about the size shown in the table. It seems like that the compression algorithms cannot compress their own results very well. Note that a single vehicle data card can never be compressed as well as several at once, since the compression algorithm (namely gzip) can only consider the redundancy contained in one vehicle data card, and it also produces overhead for each vehicle data card. The effect is that a vehicle data card occupies more than 4 times the space it would need when compressed together with other vehicle data cards.

Now, how did I get the Smart Updates smaller ? Very simple: I first uncompress the new and changed vehicle data cards found in the database and then compress them again, altogether.

5.3.5 Using Timestamp Replication

There is principally no problem in using replication based on record timestamps for FDOK. These timestamps are already present in the database. But for the reasons given in the last section, one has to uncompress the data card objects before compressing them for transmission over the network.

[6]namely zip, gzip, bzip2, and compress.

When using this approach, I would strongly recommend caching the generated updates, since the compression needs a lot of computing time on the source node.

5.3.6 Subdividing the Database for PAID

The database can principally be subdivided to contain an arbitrary subset of vehicle data cards for each node. But for the following reasons, it makes sense to find subset representations which cover a bunch of vehicle data cards:

- For newly created vehicle data cards, there must be some rules telling which vehicle data cards a node is interested in.
- To find out which vehicle data card updates are needed by a target node, the source node must know which vehicle data cards are installed there. When only naming single vehicle data cards, this information can get very large.
- When initially installing the data, there must be some useful description about what subsets are available, e.g. ’C class, europe’.
- When multicasting updates, it is useful when one can e.g. send all updates for ’M-class, USA’ to all nodes which are interested in that. Else one would have to calculate a good mixture of common subsets of vehicle data cards each time a multicast is initiated.

The last three points are currently of low interest, since only a few vehicle data cards are updated. But since it is planned to change that, they should be considered as well.

I did choose the following criteria to determine subsets:

- The vehicle model line (BR). This can be taken from the vehicle identification number (FIN). The user might prefer to choose vehicle classes instead of individual model lines.
- The area code from the order number. It was suggested to me by Mercedes that only main areas (The first digit: Germany (2), Europe (5), America (7), . . .) should be used to select the local subset.
  The problem in using a finer subdivision is that some area codes are very fine (single branches, mostly in Germany) and some are very rough (e.g. USA), which is quite confusing. For PAID, these codes may be although be used when grouping them to reflect sensible configurations.
- The build date of the vehicle (year). This is currently irrelevant, but will become more important in the future, as vehicle data cards are very seldom removed from the database.

These items might be presented to the user by PAID in a hierarchical view, to allow him to select the wanted subsets fast (’All Europe’) but as fine as he might want to. It might make sense that a user selects dependent subsets, for example a Scotch workshop selecting ’Utility vehicles, Europe’ together with ’Passenger Cars, UK’. Also, there should be some profiles reflecting preselected configurations.

Representation of Subsets

The following diagram (5.10) describes the internal representation for subsets of the FDOK database.

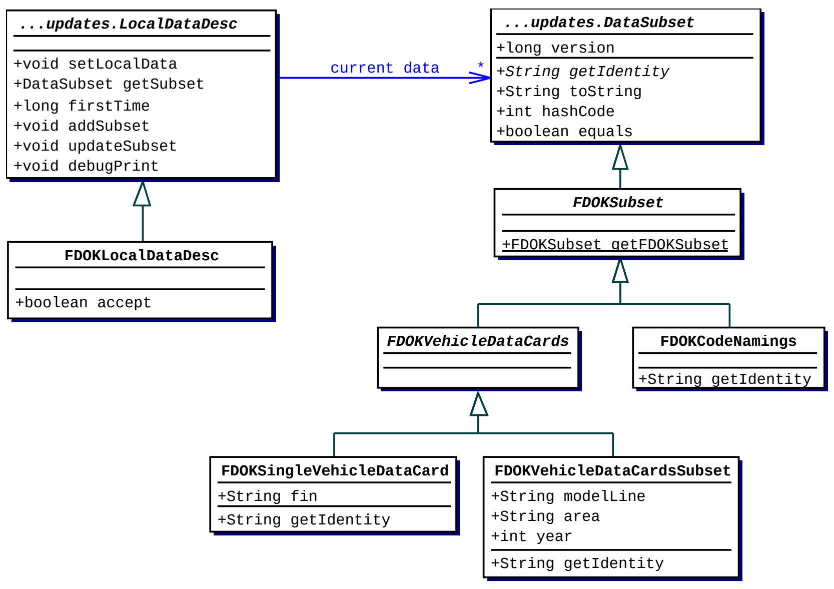

Figure 5.10: Local data description and subsets of FDOK database

The code description tables are an extra subset (FDOKCodeNamings); a subset of the vehicle data cards (FDOKVehicleDataCards) is determined by the items described above (model line, area, build year), or is a single vehicle.

The latter was included to allow PAID to learn about accessed vehicles, and to cache each accessed vehicle for some time (if the subset it belongs to is not already local). If some vehicle data cards of a possible FDOKVehicleDataCardsSubset were accessed remotely, the whole subset should be installed locally, to save computing time and to improve the ability to broadcast or multicast.

For the current implementation, specifying both FDOKVehcicleDataCardsSubset and a single vehicle data card out of that is considered to be invalid.

Chapter 6

Analyzer Design

The first section in this chapter describes how the analyzer tool is embedded into the supply processes of the StarNetwork central database StarDB and the future PAID network. After describing the subsystem decomposition and the package structure, the individual subsystems, packages, and layers are described in detail.

Not all classes and methods are described here. A complete reference can be found in the javadoc documentation accompanying this thesis.

6.1 Embedding of Analyzer Tool into Supply Processes

The following figure (6.1) shows how the analyzer tool is embedded into the supply processes of the StarNetwork central database StarDB.

The analyzer tool is placed logically between the conversion from the legacy databases and StarDB. This conversion, intermediate testing, and loading of the database is done in the usual way as described in chapter 5. But instead of being loaded into the StarDB, the converted data is put into a small temporary database. That database may already exist for EPC as the mentioned intermediate testing database. The analyzer tool is used to generate the differences between the old and new data in form of Smart Updates. These updates are stored in a stable queue, which holds them for some long time, about several months (needs to be defined). Note that there is only one queue, from which the updates are extracted and customized for the nodes which have only some subsets of the data.

The intermediate nodes in the PAID network have stable update queues, too, but they don't need to hold the updates for as long as the central database, because they can fetch some of the older updates from there when necessary.

Placing the analyzer tool after the conversion has the advantage that it doesn't have to deal with the legacy data directly, and that there was no need to re-implement the conversion. If the analyzer tool was placed before the conversion, the generated updates would have to perform the conversion when executing themselves on the database. Letting the analyzer tool operate on the databases made it also possible to follow a more universal approach for the design of the tool, as most parts of the implementation operate on relational databases in general. The generated Smart Updates are distributed and stored as serialized Java classes, and have a standardized interface to be customized and executed. Section 4.4 explains the general concepts used for these updates, section 6.5.1 shows how they are designed and implemented for the EPC database, and section 6.6.1 describes the updates for the FDOK database.

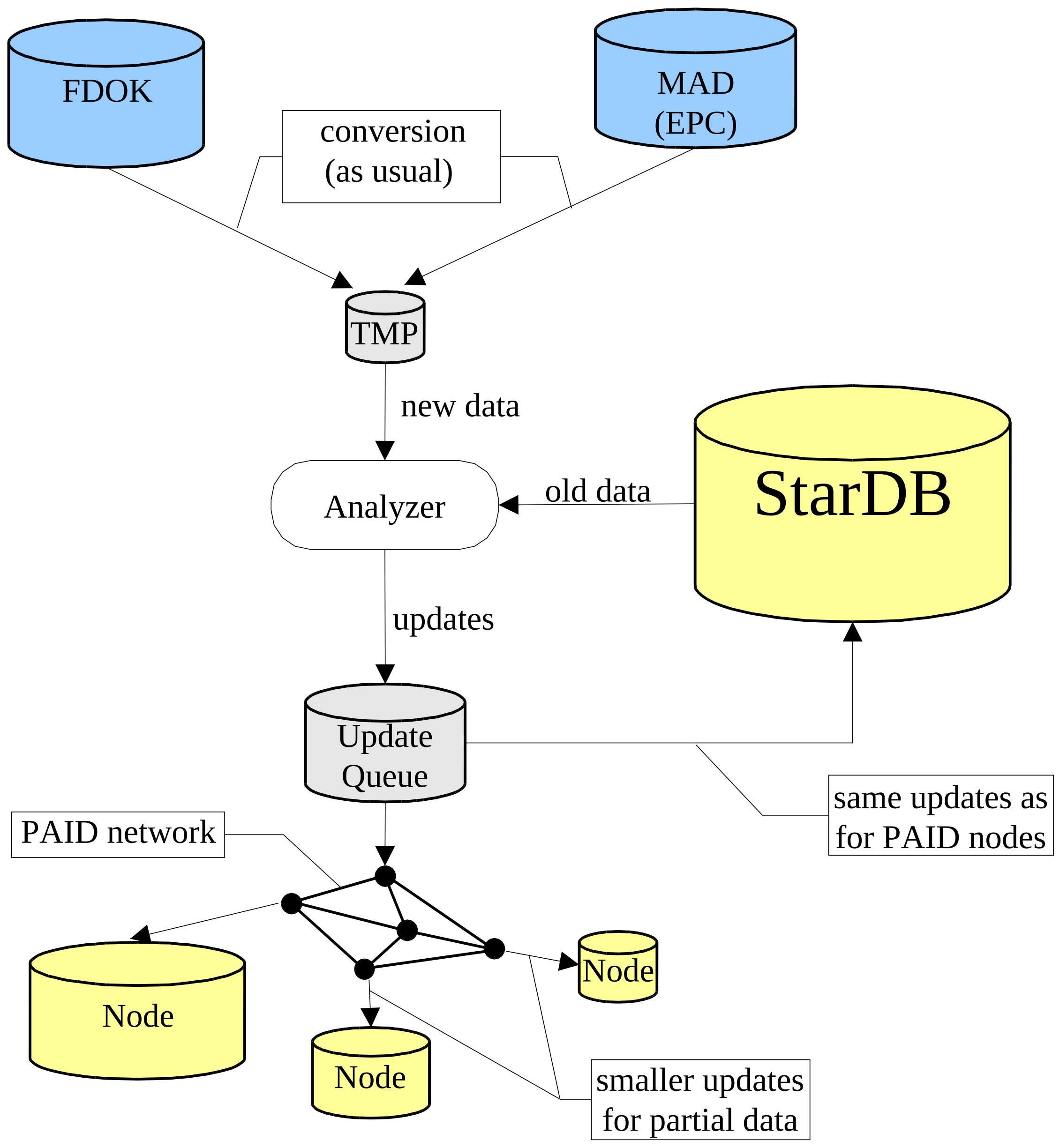

Figure 6.1: Embedding of analyzer tool into supply processes

Section 6.2.6 describes a possible verifying mechanism which makes the generation of inconsistent updates very unlikely, because the updated data is compared to the new data before the update is published. But in case there are still some undetected bugs in the change detection or update generation process, using the same updates for the StarDB as for the PAID nodes guarantees that the StarDB and PAID nodes still have the same data. After fixing the bugs, the process can continue without requiring further interaction. The inconsistent data contained in the StarDB is overwritten with the next updates generated, and those updates will also bring the remote nodes into a consistent state. No special recovery actions such as resynchronization will be needed.

6.2 Subsystem Decomposition

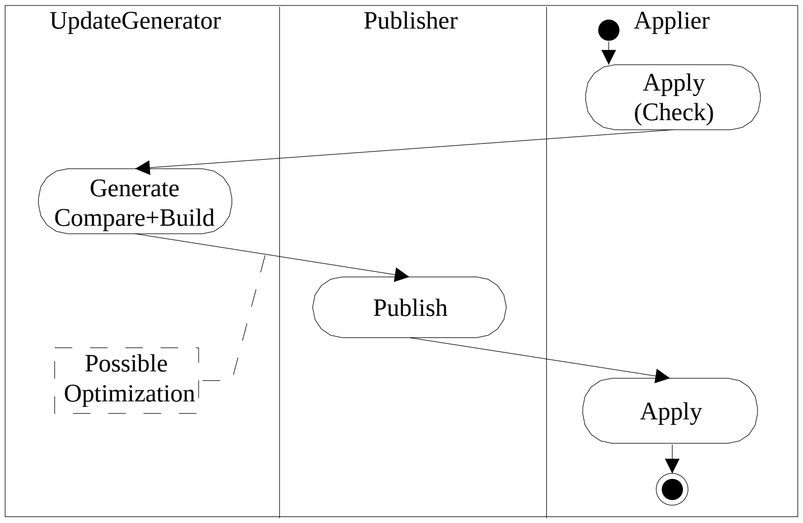

Figure 6.2: Activity diagram of analyzer subsystems

As shown in the activity diagram 6.2, the highest level of the analyzer tool is functionally decomposed into the subsystems listed below. This decomposition was just naturally derived from the functions the analyzer tool has to perform: generate updates, apply them on the local data, and publish them to make them available to other nodes. The following are short descriptions of the subsystems:

- The *UpdateGenerator* subsystem consists of the Compare subsystem and the UpdateBuilder subsystem. The former generates the differences between the old and the new data as UpdateComponents; the latter classifies, packages, and structures these into the compound Smart Updates.

- The *Optimizer* subsystem, which optionally optimizes the generated updates before they are published. Optimization is only shown as a note on the activity diagram because it was not realized. Section 6.2.2 explains why.

- The *Publisher* subsystem which is responsible for putting the generated updates into the update queue, from which the updates can then be distributed. It also determines the dependencies between the updates.

- The *Applier* subsystem. After the updates are generated and published, they are applied on the local data. This subsystem can also be used on the remote nodes to execute the updates.
  It can be confusing that apply is called before the generation of updates. This simply ensures that the reference database, which is used by the Compare subsystem, is up-to-date. Otherwise, if for some reason yet unapplied updates are in the update queue, the Compare subsystem uses an obsolete reference database, which might produce unpredictable results. Section 6.2.6 describes a smarter way to prevent the system from producing incorrect results due to bugs or failures.

- A temporary update queue, which serves as an intermediate storage (not shown). It is used between the UpdateGenerator and the Publisher subsystem.

The temporary update queue was introduced to achieve scalability: It was assumed that the updates the analyzer tool may need to handle within one invocation are too big to be stored in memory. So an intermediate disk storage was introduced. The amount of updates that the analyzer tool can handle is therefore principally unlimited.

- The public update queue, which holds the updates until most of the nodes should have received them. It was shown on the overview in the last section.

The subsystems in this approach are very loosely coupled. The update queues serve as the communication platform for the subsystems.

6.2.1 Update Generator Subsystem

The interaction diagram 6.3 shows the interaction between the Compare and the UpdateBuilder subsystem.

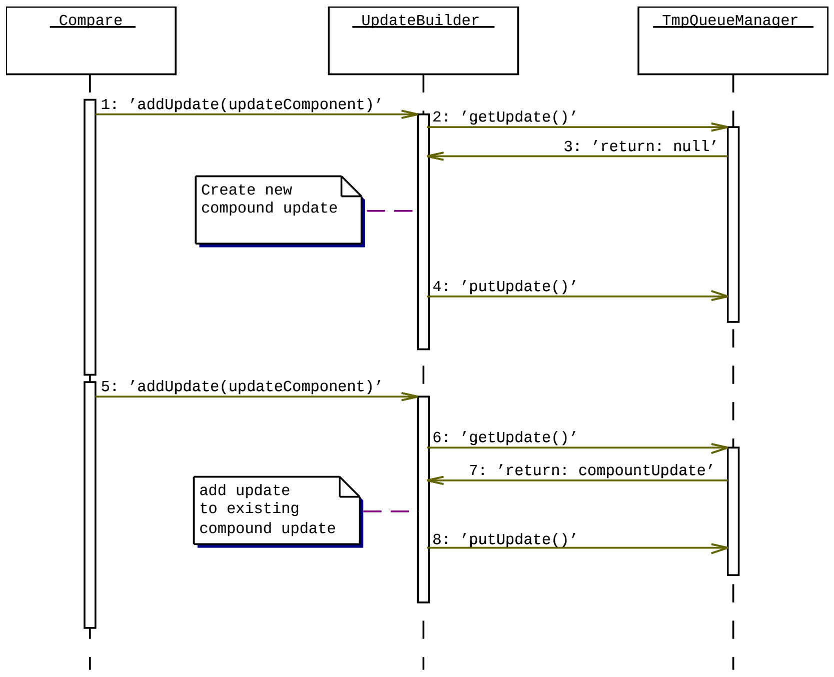
Compare
UpdateBuilder
TmpQueueManager
1: 'addUpdate(updateComponent)'
2: 'getUpdate()'
3: 'return: null'
Create new compound update
4: 'putUpdate()'
5: 'addUpdate(updateComponent)'
6: 'getUpdate()'
7: 'return: compountUpdate'
add update to existing compound update
8: 'putUpdate()'

Figure 6.3: Interaction diagram showing update generation

The compare subsystem generally compares an old with a new version of data. It then generates update operations (record updates for relational databases), which can update the old version to the new one. But these update operations are not necessarily executable in the form and order they are generated. An example for this are the record moves described in section 6.4.2. Therefore, these update operations are passed as 'UpdateComponents' to the UpdateBuilder subsystem, which combines them to produce compound, structured updates, namely the already introduced Smart Updates.

The sequence diagram 6.3 shows the typical operations when the compare subsystem generates an update component. The UpdateBuilder selects the appropriate compound Smart Update from the change queue and adds the component to that. Whenever necessary, a new Smart Update is created. The temporary update queue maintains a cache which is necessary to achieve performance.

Two general classes, Compare and CompoundCompare which follow the composite pattern were created (not shown on a diagram). The idea was to support combining several different types of compare subsystems, e.g. for databases, filesystems, etc. to one compound compare component, as the analyzer tool was originally designed for heterogeneous data. Since it turned out that actually only relational databases are to be analyzed (see section 5.2.12), this mechanism is currently unused.

6.2.2 Optimizer Subsystem

An optimizer maybe introduced to do an additional optimization of updates after their generation. It may be used for example to find some redundancy within the generated updates. It can be omitted if such optimizations don't make sense for some data, or the generated updates are already small enough.

An optimizer was initially planned for the EPC problem domain, where often similar changes occur to several or all records stored in a database table. The current database compare subsystem, which will be described later, will only generate updates to single records, which can be quite a lot.

But experiments have shown that optimization is unnecessary for the EPC updates when using the 'gzip' compression of updates together with some small local optimizations embedded into the database compare subsystem, which are described in section 6.4.3.

I learned that using such an effective compression algorithm, trying to reduce redundancy before the compression has a very minimal effect on the size of the updates. The only effective way of reducing the size of updates is to remove unnecessary information. A good example for this is the size reduction of FDOK mentioned in 5.3.3, where the unneeded serialization information was suppressed. Also, as mentioned in chapter 5 describing the EPC database, removing unneeded information for unwanted languages has a noticeable effect after compression.

Since the optimizer subsystem is neither needed for FDOK nor for EPC, I dropped the idea completely. For other databases from yet unknown problem domains, the need for an optimizer has to be figured out again.

6.2.3 Publisher Subsystem

The update publisher is responsible for making the updates publicly available, which means putting them into the update queue. It also puts the updates into the right order and determines the dependencies between them. This cannot be done by the update builder because it sees only one update at a time, and it doesn't know when it receives the last update component for a compound Smart Update from the compare subsystem.

There are no general classes available for the publisher subsystem, only classes for each problem domain. There is no general behavior of the publisher subsystem, since ordering of and dependencies between updates depend solely on the individual application domain.

6.2.4 Update Queues

The update queues are currently realized as database tables, into which the updates are stored as compressed, serialized Java objects. This is only done for this proof-of-concept prototype of the analyzer tool.

The reasons for choosing this way of storing updates was again that the PAID project wasn't launched when I started this thesis. I didn't knew which persistency mechanisms (e.g. an object database) would be chosen. I wanted to avoid introducing yet another database besides the relational databases that would be there anyhow.

This way of storing updates turned out to be a *severe* performance bottleneck since Java serialization is quite slow. For PAID, this type of update queue storage should be avoided.

The design of the updates, namely choosing strings as identifiers for data subsets, updates, and new data subsets was also affected by this decision, since I wanted these things to be available with fast access and in human readable form outside the serialized objects.

The updates are stored using a database table containing the identity, the version, the info about new data subsets, and the gzipped serialized Java object. Sections 6.4.5 and 6.4.6 describe the implementation classes of the temporary and public update queue managers, respectively.

6.2.5 Applier Subsystem

A simple apply subsystem (SimpleApply) was created mainly for testing purposes. The following things are missing in this simple implementation of the apply subsystem:

- The update dependencies are not evaluated correctly, especially dependencies between updates.
- It is assumed that the updates are delivered in a executable order by the update queue. The publisher components were implemented in a way to achieve this by ordering the updates.
- Updates are executed one by one, which means that updates depending on one or more other updates will produce intermediate, inconsistent states. If the apply process is interrupted, these potentially inconsistent states will persist.

The following sequence diagram 6.4 shows how the apply process works.

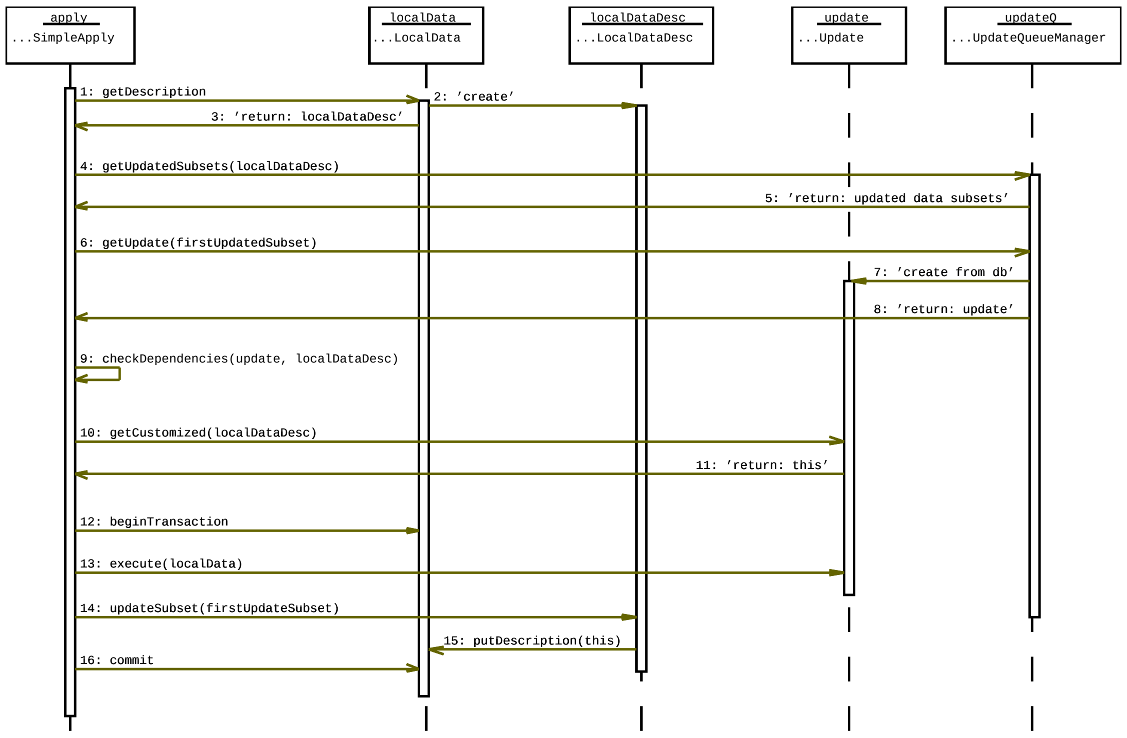

Figure 6.4: Interaction showing SimpleApply executing an Update

First, a list of available updates is requested from the update queue, using a description of the local data (LocalDataDesc) as filter. Apply then fetches the first Smart Update from the queue and checks its dependencies. If the dependencies could be resolved (which is assumed in the diagram), the update can be executed, and the description of the local data is also updated.

Executing the Smart Update and updating the description is done within one transaction, to prevent data inconsistencies due to failures.

6.2.6 Verifying Updates and Handling Failures

The following is the descriptions of an algorithm which allows to verify that no incorrect updates are distributed.

- The updates are generated in the same way as described already.
- The publisher opens a transaction to the public update queue, and puts the generated updates therein. But the transaction is not yet committed.
- The applier opens another transaction to the main database, and applies the updates. But the transaction is not committed as well.
- An additional verifier subsystem compares the main database with the temporary database containing the new data. This can be done by a slightly modified compare subsystem.
- If an error is detected, the transactions to the update queue and to the main database are rolled back. Otherwise, both transactions are committed synchronized.

This algorithm guarantees that incorrectly generated updates never get visible.

6.3 Package Decomposition

The package decomposition was done in two dimensions: One for the subsystems, and one for the problem domain. The packages for the subsystems (compare, build, apply, queues, and updates) are located in the top-level of the hierarchy and are repeated as subpackages of the application domain specific packages, which contain the relational database (db), EPC, and FDOK specific classes, respectively.

This way of organizing packages was done to force a clean separation of general and application domain specific classes. It also allows to add new packages containing classes for yet unknown problem domains (e.g. new aftersales information types) or data representations (e.g. filesystem) in the same way.

The package diagram 6.5 shows the available packages, the following list shows the complete package hierarchy along with short comments of what is contained in a package:

```
paid/datamove                   root of package tree

.../updates                     Smart Update and data representations
.../compare                     compare interface
.../build                       interfaces for update builder and update components
.../apply                       simple apply subsystem
.../queues                      interfaces for update queues
```

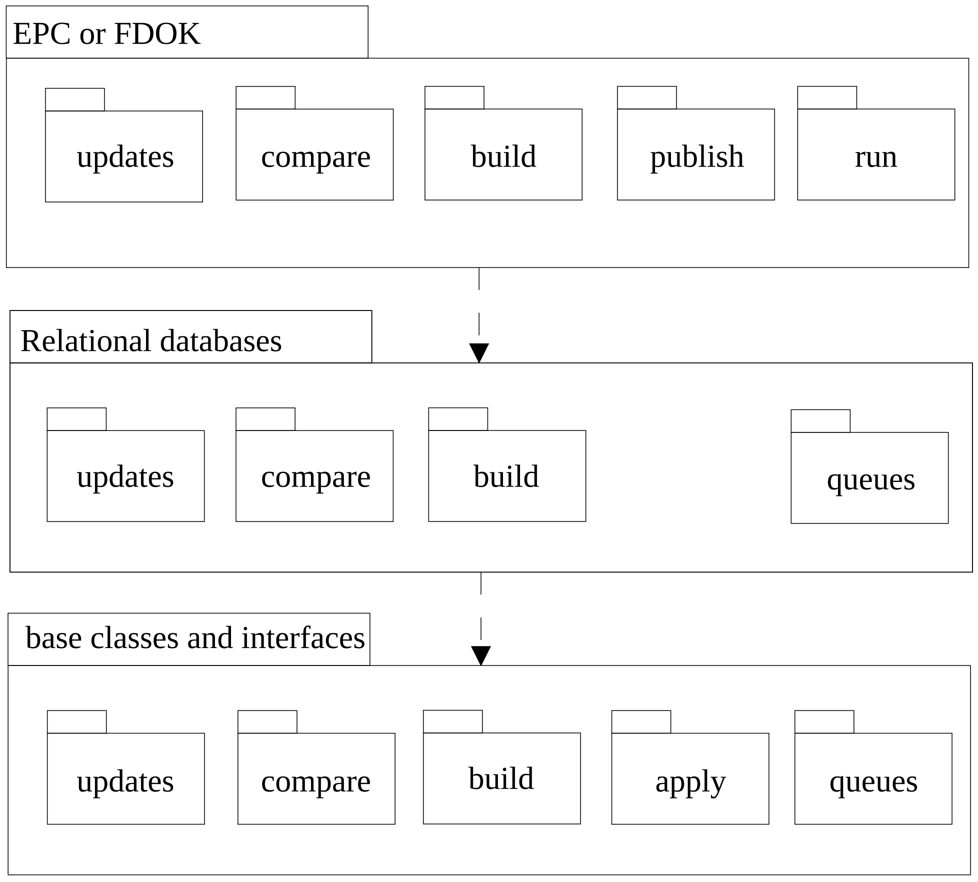

Figure 6.5: Package diagram showing package decomposition

.../db/updates	database record and table updates, LocalData impl. for db
.../db/compare	compare components for comparing relational databases
.../db/compare/util	utilities for comparison
.../db/compare/findbest	heuristic algorithms
.../db/build	database update component and its producer
.../db/queues	implementations of update queues in database
.../epc	general EPC specific classes
.../epc/updates	updates and local data representations
.../epc/build	update builder subsystem
.../epc/compare	compare subsystem
.../epc/publish	publisher subsystem
.../epc/run	classes containing main()
.../fdok	general FDOK specific classes
.../fdok/updates	updates and local data representations
.../fdok/build	update builder subsystem
.../fdok/compare	compare subsystem
.../fdok/publish	publisher subsystem
.../fdok/run	classes containing main()
.../db	database abstraction layer
.../db/columns	implementations for column types
.../db/values	implementations for value types
.../db/constraints	implementations for constraints

6.4 Support for Relational Databases

This section describes all classes which are specific for relational databases. This is the biggest part of the analyzer in terms of number of classes and lines of code. The sub-sections that follow will describe:

The database abstraction layer as an intermediate layer above the JDBC access layer. It provides an abstraction of the entities (tables, columns, values, constraints) found in the database. It is also used to hide the differences of the individual database management systems.

The database updates which are detected by the database compare components: These are updates to database tables and the record updates they contain.

The database compare components which are used to build up the application specific compare subsystems. Different algorithms are provided which can be used or extended to compare database tables.

The database update component delivered to the update builder subsystem by the database compare components.

All classes described within this section are located in or below the package ’paid.datamove.db’. By convention, all names of database specific classes start with ’DB’, which means database, not Daimler-Benz.

6.4.1 The Database Abstraction Layer

This layer is very different from the “normal” relational database layers available for Java. The commercially available database utilities are often used to provide a mapping between classes and the relational database representation. You already have designed some classes, and use some tool to automatically generate the appropriate database definitions and the classes used to store and retrieve instances of the designed classes. The requirements for this database abstraction layer were the other way round: I have some database schema, and want to automatically and dynamically detect what tables and attributes are used therein. The precondition for this is a JDBC driver which makes that information available to a Java application, which most drivers do.

The database abstraction layer was designed mainly to support the updates and compare algorithms described in the next sub-sections, but it can also be used in other ways, as for example to execute arbitrary SQL queries, or to store serialized Java objects. The latter is currently used by the update queue managers.

It is assumed that the reader is familiar with relational database concepts and the standard JDBC API [JDBC98].

The UML class diagram (6.6) shows the main classes found here. They were designed and named closely to the typical objects used in the context of relational databases: tables, columns, and values. The next section contains a typical usage example. The following are explanations of the classes shown on the diagram.

Note that most object attributes are publicly accessible but declared as ’final’ to protect them from being changed by other objects. In fact, most of the objects cannot be changed after they are created.

DBConnection is a wrapper for java.sql.Connection. It represents a connection to a database, and is constructed by giving the JDBC URL of a database combined with the schema name and the

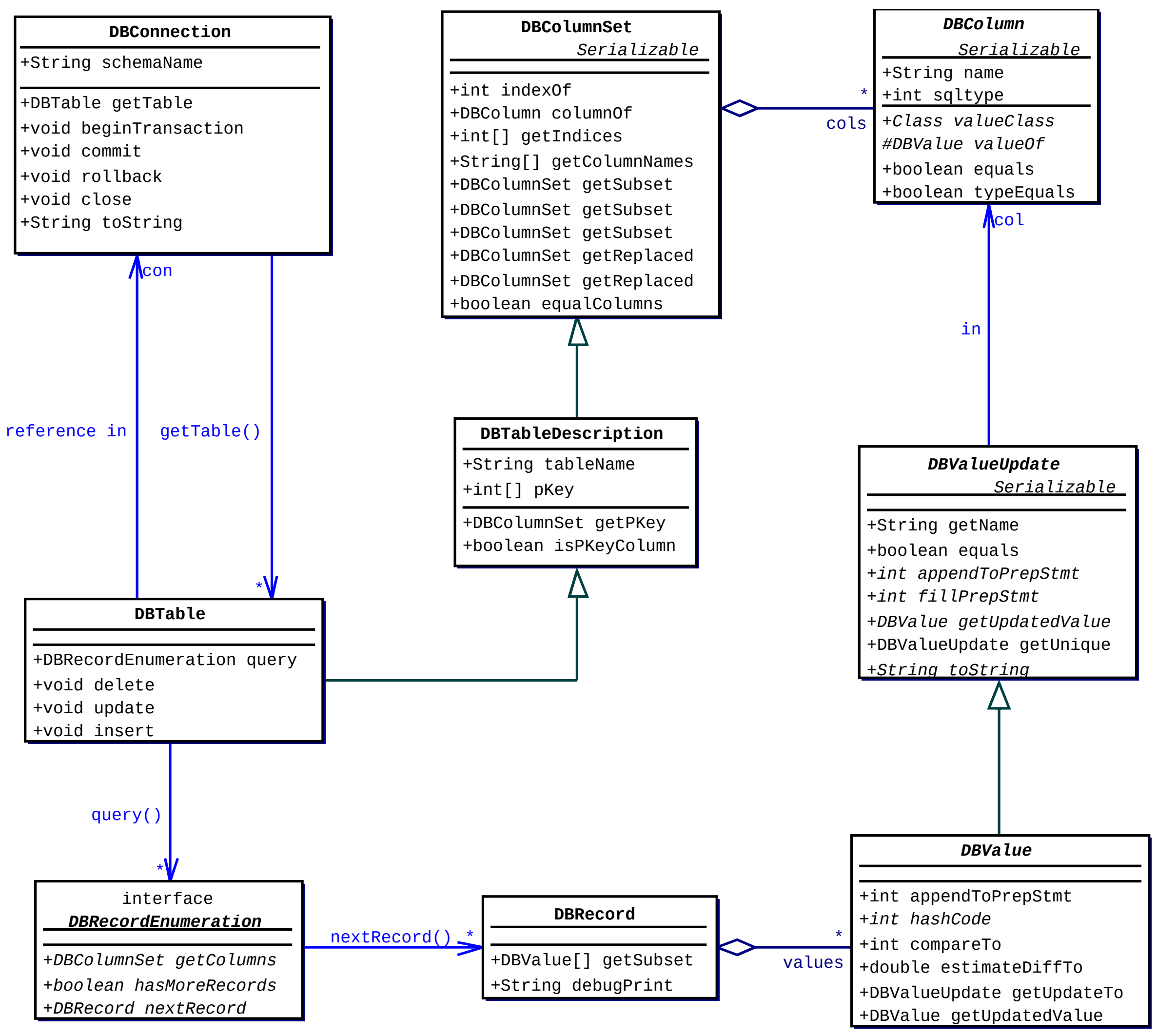

Figure 6.6: Class diagram of database abstraction layer

user and password used to access the database. It also contains a link to a helper class (described later), which is used to make operations database independent. JDBC provides many database independent concepts, but you still have to use SQL commands, which are not completely standardized.

DBColumn is an abstraction of a database column. It contains the name and the type of a column found in the database. It also provides the method 'valueOf' used to produce the appropriate DBValue objects from a query. That method is implemented by the subclasses of this abstract class. This mechanism provides a possible extension of handling special attribute types not covered by the standard JDBC types or the already implemented DBColumn subclasses. The section 'Columns and Values' provides more information and examples of subclasses.

DBColumnSet is an ordered set of columns. It contains a publicly accessible array of DBColumn objects, and a hidden hashtable which allows fast access to columns contained in the set. Methods are provided to access the columns by name, produce subsets of columns, or generate new column sets by replacing column handlers (DBColumn) with customized or more specialized ones.

DBTableDescription represents the automatically detected set of columns in a database table, and describes the table's name and primary key. The latter is just an array of indices of the primary key columns. This class is only a description of a table, not a reference to a database table itself (which is DBTable). The reason for separating the description and the actual reference is that the description may refer to more than one table. This is used for example by the compare algorithms, where such a description refers to an old and a new version of a table contained in different databases. Also, a table description is serializable which wouldn't make sense for an actual table reference.

DBTable is the actual reference to a database table, which is also its description. It can be used to query, update, or delete the records contained in that table or to insert new ones. The only way to construct DBTable objects is by calling getTable() on a connection to a database with the name of a table contained in that database. That method makes use of a helper class DBTable-Factory (not shown), which fetches and converts the necessary information about columns and the primary key from the database meta data.

DBRecord represents a set of values. The array containing these values as DBValue objects is publicly accessible. If the record was produced by a query, it is guaranteed that the array corresponds to the DBColumnSet used as argument for that query. Normally, a record represents a row in a table, but it might also be produced by a query to multiple tables. Support for these was taken into consideration but is not implemented.

DBRecordEnumeration is an interface for getting records from some kind of source. The standard implementation (DBRecordEnumerationImpl) delivers the records returned by a query. There is a possibility of manipulating the records returned by a DBRecordEnumeration described in the section 'Simulation of Record Changes'.

DBValue represents the value in a single field of a table. It provides methods for comparing it to other values, generating and estimating the differences to other values, and for writing the value to the database. The section 'Values and Value Updates' provides more information.

DBValueUpdate represents an update to a value in the database. In general the updated value can only be determined if the previous value is known. Since every value can be updated by overwriting it, each DBValue object is also a value update.

A Simple Example

The sequence diagram 6.7 shows typical sequences of method calls for a common usage of the database abstraction layer.

The example shows the sequence of method calls used by an arbitrary acting object which wants to get some information (a debug string for simplicity) from the first record returned by a query. To obtain this it first creates a connection to the database. After that, it gets a reference to a table therein and queries it. The acting object asks the record enumeration returned by that query if it contains any record (which it does), gets the first record, and request the debug string from that record.

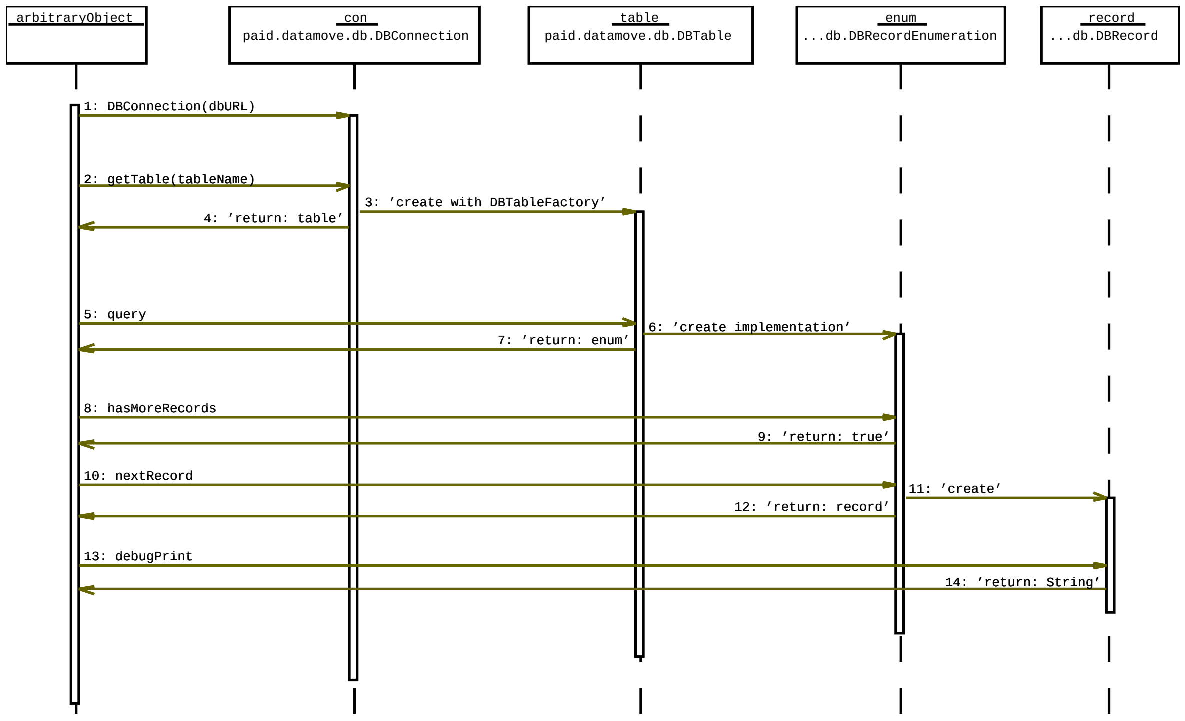

Figure 6.7: Typical sequence of method calls for a query

Columns and Values.

This section explains the interaction between columns and values. DBValues are ’produced’ by the column they are contained in. ’Producing values’ means that if an instance of DBColumn is contained in a query’s column set, the records delivered by that query will contain the ’produced’ types of DBValue at the corresponding positions.

The class diagram 6.8 shows only some examples of DBValue classes. The DBAutoColumn produces many more DBValue subclasses than shown on the diagram, since each standard JDBC type has a corresponding DB. . . Value (for example: Byte, Short, Int, Long, Double, Binary, Date, Time, . . .). Which value is produced in particular is determined by the ’sqltype’ the column was created for. A table reference which is returned by DBConnection.getTable() will contain only this type of columns. The denotation ’auto’ expresses this fact: This column type is created automatically.

The DBObjectColumn can be used for database columns which store (gzip-)compressed serialized Java objects. So it produces only DBObjectValue and DBNullValue objects. The next section contains an example showing how an automatically created column is overridden by this column type.

Since information about their type and column is needed, NULL values stored in the database are represented by DBNullValue objects and not by Java ’null’ values, as one might expect.

The concrete implementations of the abstract DBColumn class are located in a separate subpackage called “columns”.

Column Sets

The following sequence diagram 6.9 shows an example of getting a customized column set. In that example we want to retrieve some java objects stored in the database. We don’t want to query values

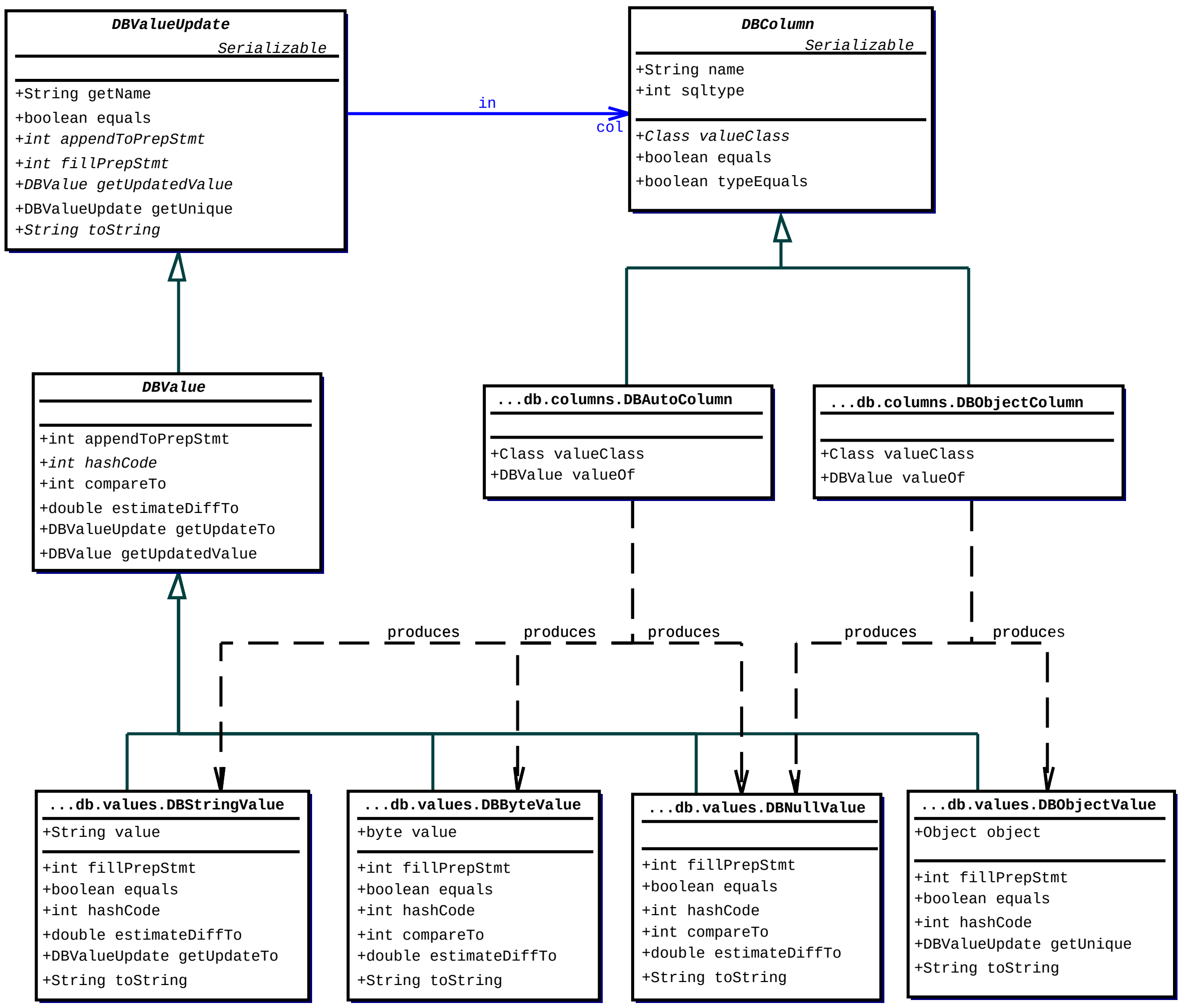

Figure 6.8: Class diagram showing database columns and produced values

from all columns contained in the table, and the column containing the objects are handled by default as if they would contain some arbitrary binary data. So we have to manipulate the default column set from the table to retrieve the serialized objects from the database directly as object references.

Since the table contains some columns we are not interested in, we first create a subset of the columns from the table, containing only the object identifiers and the objects themselves. The object column is auto-detected as a binary column, so we create a DBObjectColumn based upon that binary column. With that we create a new DBColumnSet containing this column instead of the auto-detected one. After that we query the table and get the first record the same way as described in the first simple example (not shown in this diagram).

As mentioned earlier, it is possible to extend the already implemented DBColumns and DBValues by customized ones. This is used e.g. for the FDOK data, where the values stored in the database are compressed, and a special column type was implemented which uncompresses the query results before producing standard DBBinaryValue objects (see section 6.6.3).

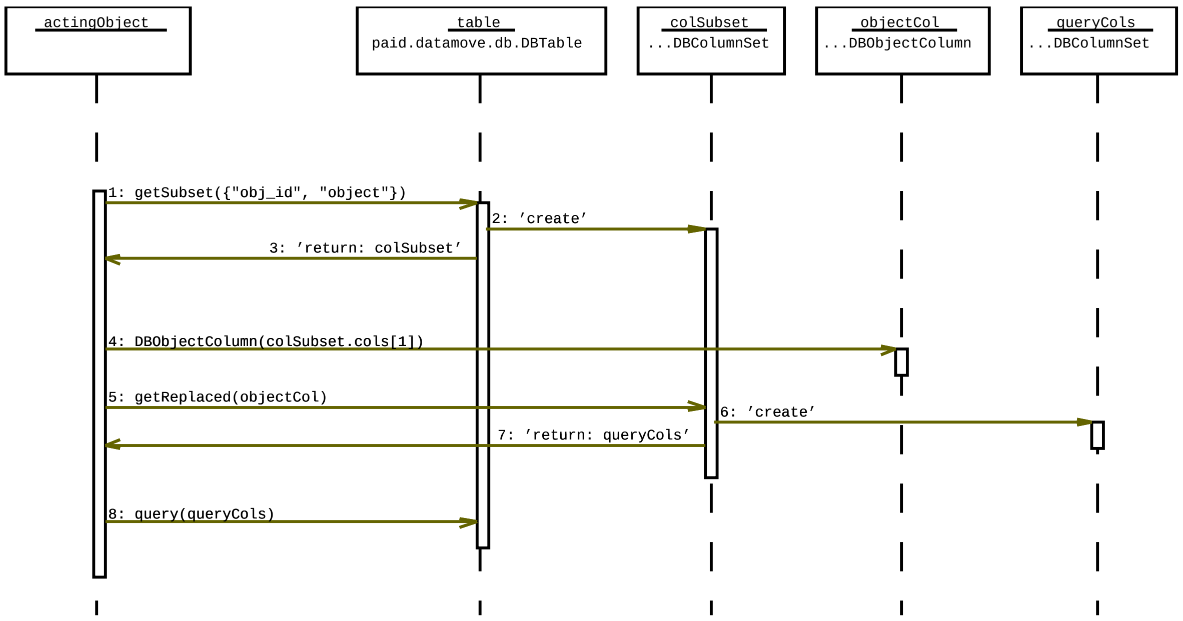

Figure 6.9: Message sequence showing customization of column set.

Values and Value Updates

DBValue and DBValueUpdate represent values respectively transformations of values in a column from a database table. They are always bound to the column they are contained in, so values from different columns are never considered to be equal.

The DBValue and DBValueUpdate are only abstract classes which define some methods the subclasses containing the individual data types have to implement. The values read from the database which are contained in the subclasses of DBValue are publicly accessible but cannot be changed. Therefore, a DBValue is always constant. The 'appendToPrepStmt' and 'fillPrepStmt' methods are used to write values (back) to the database. To understand their meaning and usage, one should be familiar with the prepared statement concept used by JDBC. The former method appends the necessary placeholders and SQL statements to a prepared statement string, and after the statement is created, the latter method puts the actual values into the placeholders.

The sequence diagram 6.10 shows the interaction between DBValueUpdates and DBValues. The example uses string values, since these are currently the only ones for which value updates are implemented.[1]You can think of a value update as a relative value: If you know the old value, a value update gives you the new (updated) value. Since you can always update a value by overwriting it, a DBValue itself is also a DBValueUpdate.

To explain this further: in the example shown, the old string could be e.g. “Hello Qorld”, and the DBStringValueUpdate could be 'concat((left(value, 6), “W”, right(value, 4))', expressed in SQL. Currently, all string value updates have this format. This means that the current generation and representation of string value updates just consist of the length of the unchanged prefix, the changed subset in between, and the length of the unchanged suffix.

Other non-trivial methods of DBValues and DBValueUpdates will be explained in the following

[1]The idea behind a value update is making the serialized updates smaller for big values. This doesn't work e.g. for numeric values, and simply wasn't implemented for binary values yet. So these values are always simply overwritten, which means the 'getUpdateTo' method just returns the new DBValue.

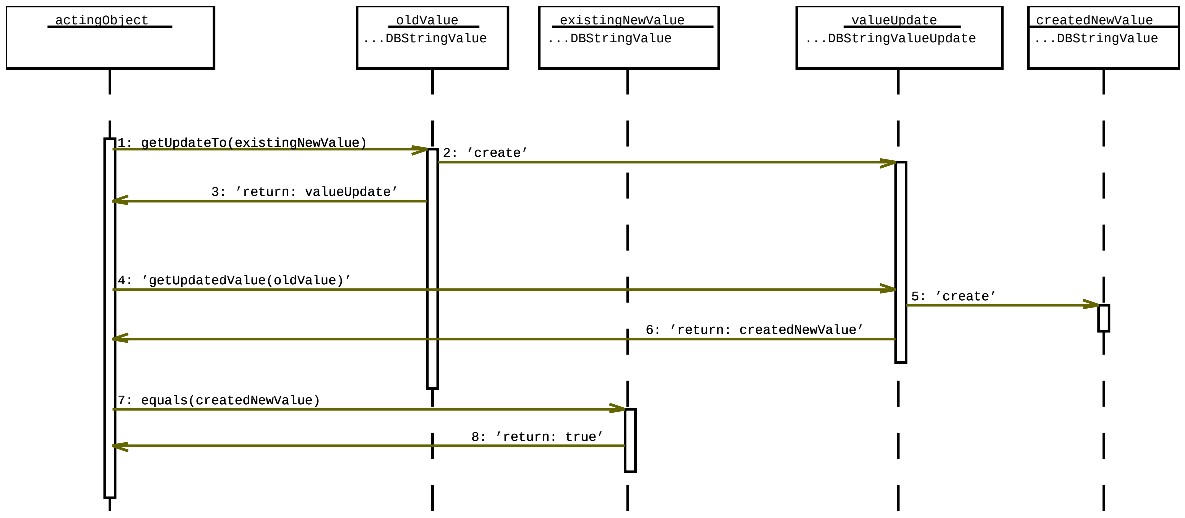
actingObject
oldValue ...DBStringValue
existingNewValue ...DBStringValue
valueUpdate ...DBStringValueUpdate
createdNewValue ...DBStringValue
1: getUpdateTo(existingNewValue)
2: 'create'
3: 'return: valueUpdate'
4: 'getUpdatedValue(oldValue)'
5: 'create'
6: 'return: createdNewValue'
7: equals(createdNewValue)
8: 'return: true'

Figure 6.10: Simple message sequence showing interaction of values and value updates.

sections: 'compareTo' and 'estimateDiffTo' are used for comparing databases, and 'getUnique' is an optimization for serialization which avoids serializing the same value twice.

The concrete implementations of the abstract DBValue and DBValueUpdate classes are located in a separate subpackage "values".

Constraints

Constraints are used for query, update, and delete operations to the database. They specify which records are affected by an operation. Class diagram 6.11 shows the classes used to construct constraints. The compound DBAndConstraint follows the composite pattern.

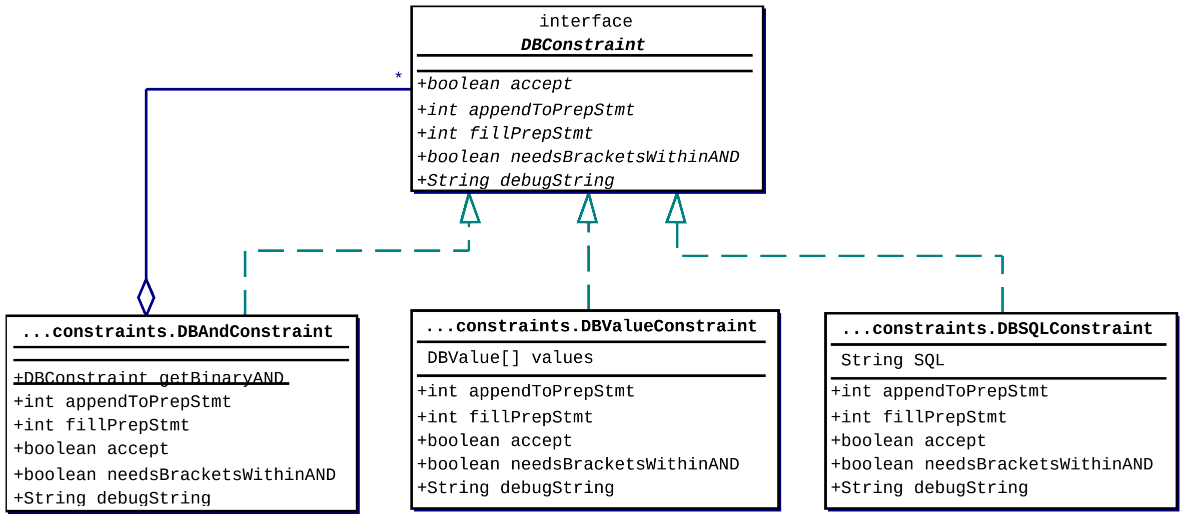

Figure 6.11: Constraint classes used for database queries, updates, and deletes.

A null value as a constraint reference means 'no constraint' by convention. A DBValueConstraint selects all records which include the values contained in that constraint. DBSQLConstraint is an arbitrary constraint formulated in SQL.

Normally, constraints are used only in statements executed by the database engine, but there is also a method ’accept’ which tells whether a DBRecord would be accepted by that constraint. This functionality certainly doesn’t work for constraints formulated only in SQL; one would have to subclass DBSQLConstraint and implement that method consistently to achieve that.

The methods ’appendToPrepStmt’ and ’fillPrepStmt’ of DBConstraint are used the same way as that of the DBValue[Update] class. The former appends the constraint expressed in SQL to a prepared statement string, and the latter fills in any necessary values after the statement was created. Depending on the JDBC driver, performance of prepared statements is currently not optimal; a cache for prepared statement might be useful.

The implementations of the DBConstraint interface have been put into a separate subpackage named “constraints”.

Support and Optimizations for Comparison and Updates

Some of the mechanism described above are designed especially for comparing databases and for efficiently generating small serialized updates. The following is a list of some of the mechanism and optimizations serving these purposes.

The ability to serialize some objects from the database abstraction layer is only used to include those objects in compound Smart Updates which are transmitted and/or stored persistently.

Value updates were designed solely for the purpose of reducing the size of updates. Since this makes only sense if the update representation is smaller than the updated value, this is only implemented for string values, as already mentioned.
Another effect is, that if many values in a column were updated the same way, the update may need to be serialized only once within a compound Smart Update. The precondition for this is that all updates are expressed equally. An example for this might be inserting a string into all values contained in a column. Transmitting all the changed values would be more expensive than transmitting the change itself. At least, even if equivalent updates resulting in different values are serialized more than once, they can be compressed better than the new values themselves.

The method getUnique() implemented by DBValues respectively DBValueUpdates returns unique instances. This ensures that equal values in a column always result in the same DBValue object. Respectively, equivalent updates are represented by the same object, as already mentioned above.
Since the compound Smart Updates the values are contained in will be compressed, there is only a little gain in space. But as experience has shown, there is a significant decrease of (de-)serialization time. The time it took to deserialize large updates containing many DBValue objects decreased from several minutes (!) to a second.
One drawback is that the mechanism of generating unique values is currently implemented by a single hashtable which never forgets values and therefore tends to grow very large. A usage count per getUnique() call and/or a LRU algorithm would solve this problem.

The DBValue.compareTo() method compares records in terms of a sort order equivalent to that used by the database engine. This is used by the database compare algorithms.
This way of comparing is not always possible, as e.g. the order of strings generally can’t be

determined. Section 6.4.3 illustrates the reasons for this. So, if two values can't be compared, a DBUncomparableException is thrown.

The DBValue.estimateDiffTo() method is used solely by the heuristic compare algorithms. As the name suggests, it returns an estimated difference between two values, which is expressed by a double value. If and only if two values are equal, '0.0' is returned. But note that using the 'equals' method is the preferred and faster way of checking whether two values are equal. As already mentioned, DBValues are only considered equal if they were produced by the same DBColumn.

SQLHelper

The following simple class diagram 6.12 show the SQLHelper class associated with the database connection DBConnection.

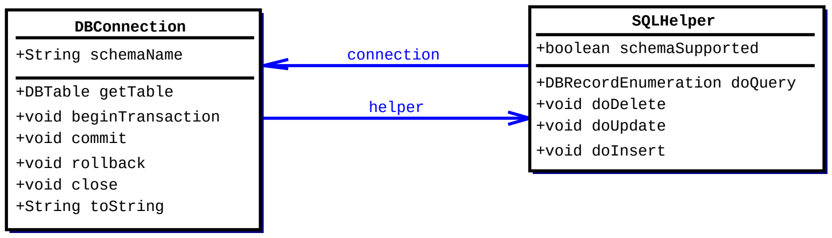

Figure 6.12: Helper class SQLHelper for producing SQL statements

The SQLHelper class was introduced to concentrate all SQL statements generated within the database abstraction layer within one place. Adapting the database abstraction layer to an incompatible database system can then be done by subclassing the SQLHelper and modifying DBConnection to determine the appropriate SQLHelper subclass based e.g. on the JDBC URL. That avoids introducing bugs into the code used for other database systems as there is no need to change that code.

Simulation of Record Changes

This section describes a mechanism built for the optimizer subsystem, which would allow an optimizer subsystem to simulate record updates.

There is support for a simulation of record changes built into the database abstraction layer. It was initially designed to support simulation of deletion and moving of records. This means hiding records respectively changing the primary key of records delivered by a record enumeration, and was named 'remap'. The purpose was to allow an optimizer subsystem to simulate the delete and move operations occurring logically before the update operations, as described in the 'Database Table Updates' section. But since the idea of the optimizer subsystem was dropped, this mechanism is currently not used.

Class diagram 6.13 shows the classes and interfaces used for remapping records. An object which wants to change the records delivered by a record enumeration has to implement the DBRemap interface and wrap a DBRecordEnumRemapImpl around the source DBRecordEnumeration.

6.4.2 Database Updates

In this section, the implemented database update operations are described. The concept used here is that a database schema update is a collection of table updates, which consist of atomic update op-

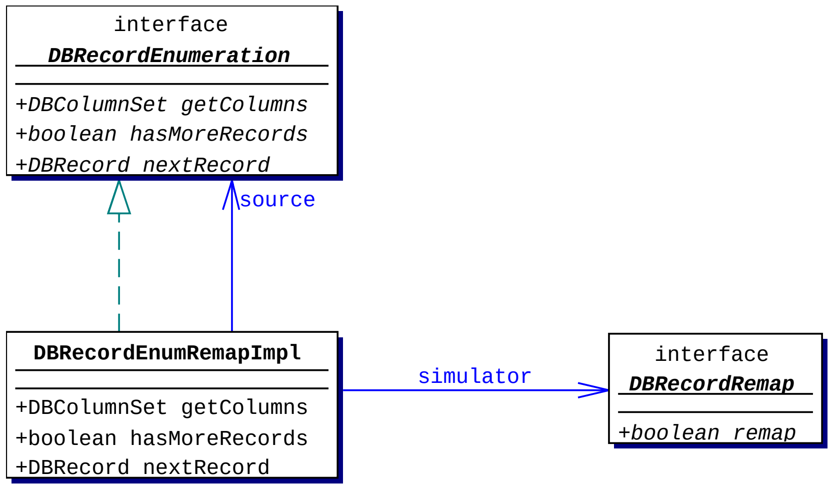

Figure 6.13: Classes and interfaces used for record change simulation

erations, namely INSERT, UPDATE and DELETE operations. The database schema update itself is neither found in this section nor in the paid.datamove.db.updates package it describes, but is implemented in the problem domain specific packages as the Smart Updates. This is because there is no general rule describing which tables are contained in a database schema…

The classes described in this chapter are located in the package ’paid.datamove.db.updates’.

Atomic Record Update Operations

The atomic record update operations represent the standard SQL record update operations. The interface DBRecordUpdate is implemented by all atomic record updates. ’Atomic’ means that the update is expressed as a single SQL INSERT, UPDATE, or DELETE statement. The implemented operations are:

Single record updates, which affect only a single record: deleting, inserting, updating, and moving records. Moving a record means changing its primary key values and is a specialized form of an update operation.
Single record updates implement an interface (DBSingleRecordUpdate) which allows some kind of introspection by querying values from the primary key of the record the operation will affect.

Multi record update operations, namely deleting and updating some records which match a given constraint. The standard compare algorithms available do not generate these operations yet. The DBAbstract…Records operations implement the common behavior of single and multi record updates.

Diagram 6.14 shows the interface and class hierarchy of the update operations.

The main method implemented by an update operation is ’execute’, which is given a reference to a database table the operation should be executed in.

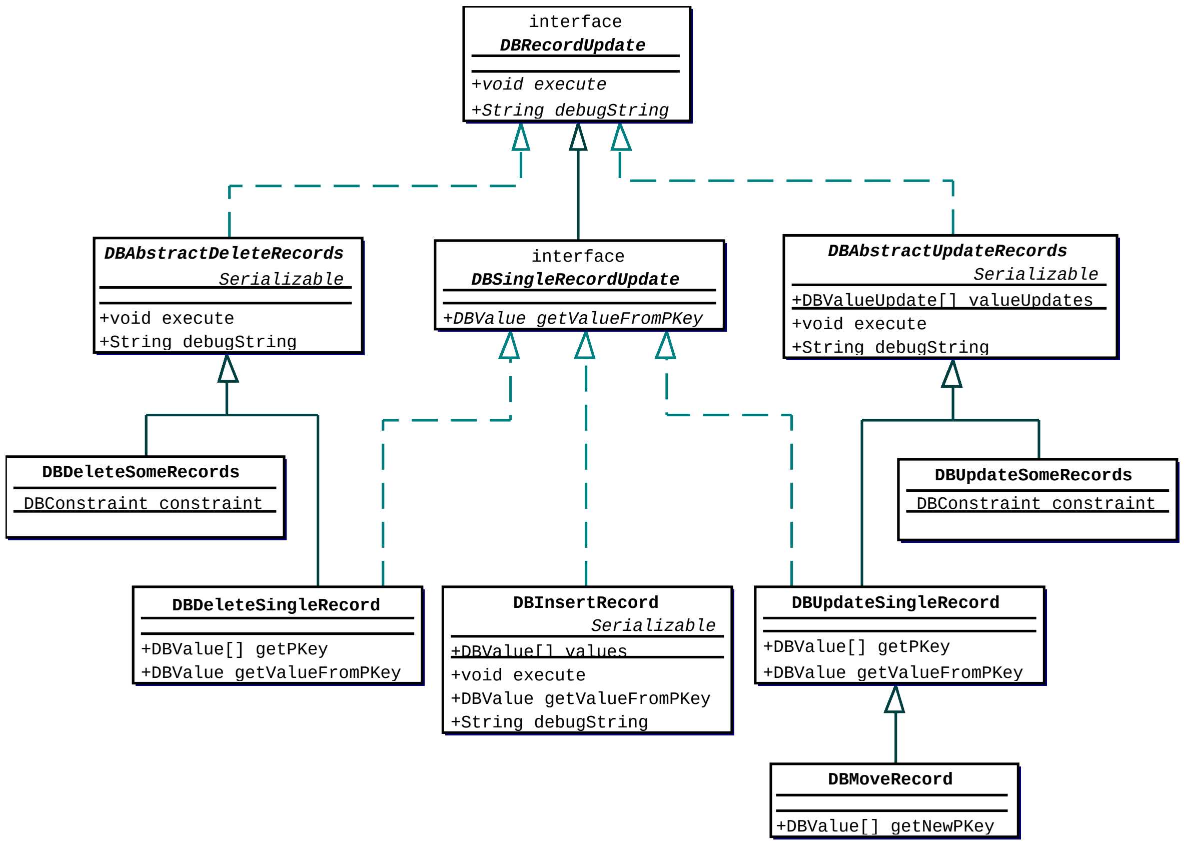

Figure 6.14: Class diagram of database record update operations

Table Updates

These are mainly containers for update operations. As shown on figure 6.15, there are two types of table updates. DBSimpleTableUpdate contains only delete, update, and insert operations; DBTableUpdate adds the ability to move records, which means changing values in the primary key of records.

The operations are executed in the following order: delete, move, update, insert. It is important to define this order since it affects the results of the operations. This execution order was developed out of the following constraints:

- Deletes must be executed before inserts to avoid possible primary key conflicts.
- It is more efficient to execute deletes before updates and updates before inserts (there are fewer records that are updated).

A consequence of this is that the primary key of DBAbstractUpdateRecords operations refers to the new primary key of moved records. The next section explains among other things why move operations have been put into a specialized subclass of DBSimpleTableUpdate.

DBSimpleTableUpdate provides a method visit(), which allows a class implementing the DBSingleRecordUpdateVisitor interface to visit all record updates it contains.

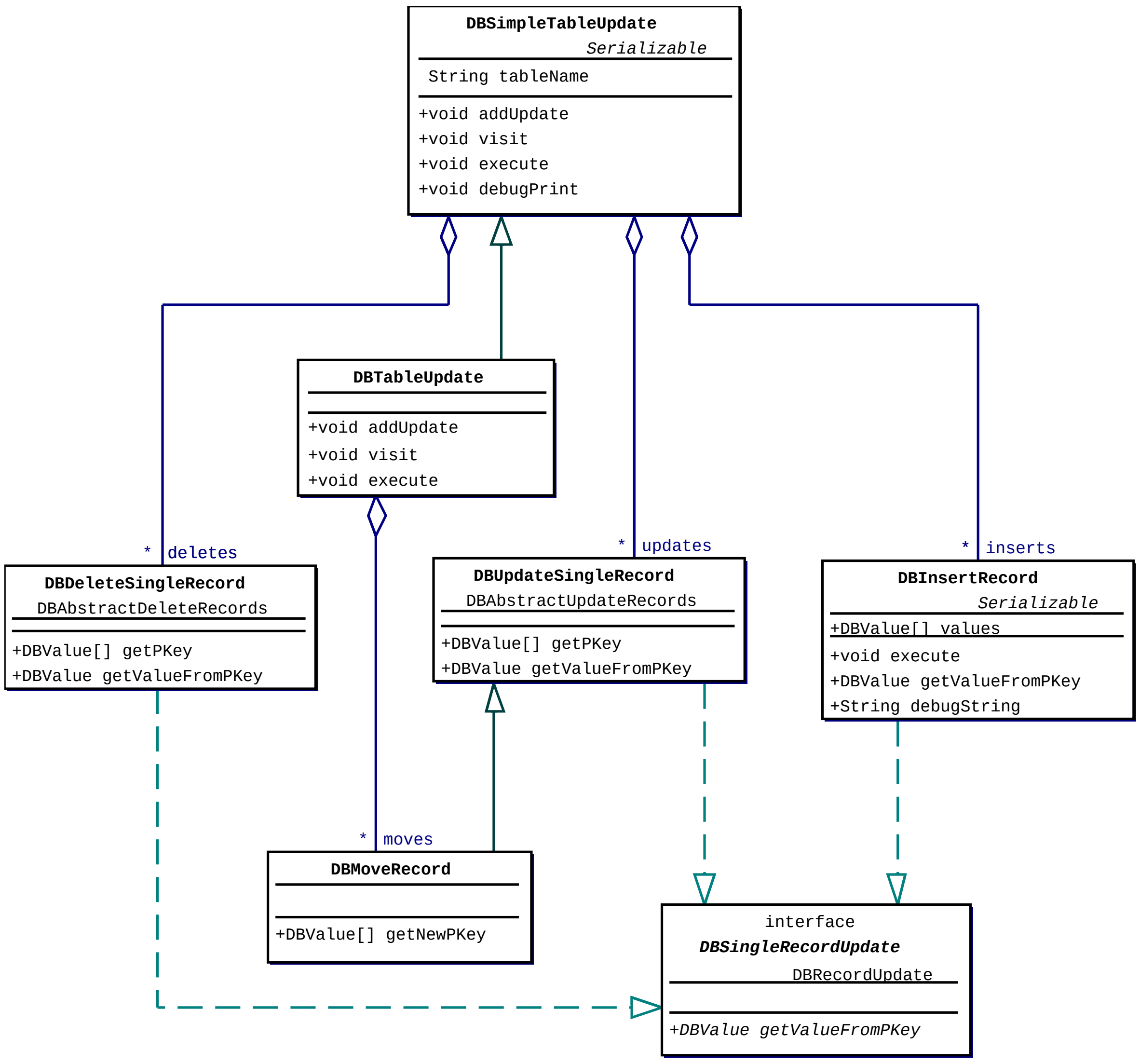

Figure 6.15: Class diagram of database table update

Moving Records

There is a problem with changing the primary key of records, consider the following example[2]: 2->1, 1->2. If you would try to execute the two updates in this example directly as SQL UPDATE operations, the first would fail with a 'duplicate key' error, because there is already a record '1'. Turning off the primary key constraint wouldn't solve the problem, because the second operation would then affect both records (think about that). This problem is equivalent to swapping the values of two variables: you need a temporary storage. In our case, this is an unused primary key value, which is given to DBTableUpdate at construction time. Because moves don't occur in every table, DBSimpleTableUpdate was introduced which doesn't need that value.

The following is the algorithm used to execute moves:

[2]The examples given in this section assume that all records are addressed only by a single number.

1. All moves are initially put into a hashtable, to allow random access to them. The key used to access the moves in that hashtable is the old primary key of the moves.
2. If the hashtable is empty, we are finished
3. An arbitrary move is taken from the hashtable and put into a (yet empty) chain of moves. This chain will then look like “1->2, 2->5, 5->3,. . . ” .
4. If the target of the last move in the chain is found in the hashtable, the corresponding move is removed from the hashtable and appended to the chain. This continues until the target of the last move in the chain wasn’t found in the hashtable.
5. If the chain forms a loop, which means the target of the last move in the chain is the source of the first, the chain is broken up using the temporary key value, e.g. the chain “1->2, 2->5, 5->1” would become “tmp->1, 1-2, 2->5, 5->tmp”.
6. The chain is executed in reverse order and cleared. Continue with step 2.

Note that this algorithm only works if the move operations make sense. Moves forming e.g. the following chain: “1->2, 2->3, 3->2” are invalid. Such loops “in the middle” of the chain generally don’t make any sense, since the result is not uniquely defined.

6.4.3 Database Compare Components

This section describes the database compare components which can be used to build up the customized compare subsystems for the problem domains. The task of the compare subsystem is to generate minimal operations which transform an old version of data into a new version.

The main non-functional requirements for the design of the database compare components were that they should be extensible and scalable. The former means that there should be many possible ways to add new functionality to allow the analyzer tool to be used on databases in yet unknown problem domains. The latter is a very important requirement since the amount of data that should be compared is not known in advance. A consequence of the scalability requirement was that nothing is held totally in memory. The database compare components operate on DBRecordEnumeration sources that deliver the records like streams.

The different algorithms and heuristics described here can be used in two ways: First, they can be plugged together in the most suitable way and customized using their parameters. Second, they can be extended by new algorithms and specialized versions of existing ones.

Some of the compare methods can be used very straight forward, e.g. simply using DBCompareMethodIterating to iterate over the records contained in the old and the new table. Others have many different parameters and are quite complicated to use.

The class diagram 6.16 shows the database compare components. The next section contains a small example code fragment and sequence diagram which will demonstrate their usage.

On the toplevel there are one or more DBCompareSchema instances which just use different DBCompareTable objects for each table the schema contains. DBCompareTable implements the interface DBRecordSource, which means mainly that it implements two methods which deliver the records from the old version and the new version of the table, respectively.

For each table, a compare method is defined. This compare method can either be one of the classes shown on the bottom, or one of the two partitioning classes. The former represent methods of comparing records, each explained in one of the following sections. The latter divide the table up

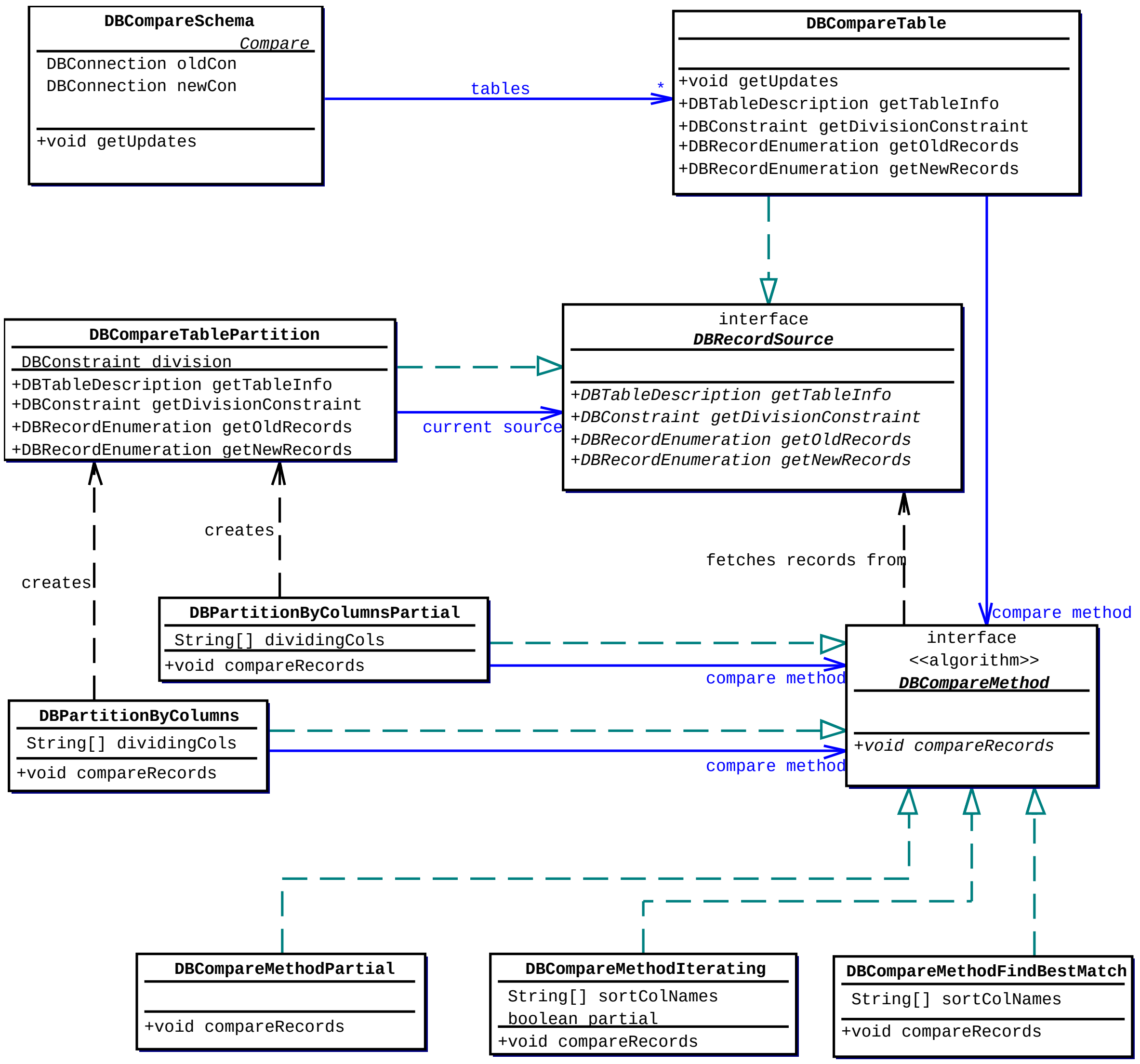

Figure 6.16: Database compare components

into partitions, and create a new record source DBCompareTablePartition for each partition. Each partition is then compared separately using the compare method assigned to the partitioning method. These mechanisms build up an principally unlimited chain of partitioning methods, each creating smaller partitions, until one instance of DBCompareMethod... is used to compare the records in the last, finest partition.

In general, there a two reasons for creating partitions. The first is that the new version of the table might include only a subset of the data contained in the old (for example, only a superset of new and changed data). The DBPartitionByColumnsPartial method can then be used to determine which partitions are available in the new version and compare only these partitions. The second reason is that some compare algorithms might have a non-linear complexity, so dividing the table up into small partitions guarantees scalability. Currently, DBCompareMethodFindBestMatch is the only compare method where this might be the case.

Simple Example

An imaginary example: We are getting data about vehicles that had built-in deficiencies which were not detected before they were sold. This data should be loaded into a consolidated database holding data from all factories in the world. The data is organized by country, company, factory, and a vehicle identification number.

The following table (6.1) shows the imaginary ’deficiency’ database table definition.

country	company	factory	vehicle	built date	deficiency	...
“USA”	“GM”	...	123..01	11/11/98	...	...
...	...	...	...	...	...	...
“Germany”	“BMW”	Munich1	...	...	...	...
...	...	...	...	...	...	...
...	...	...	...	...	...	...

Table 6.1: Imaginary Example of ’deficiency’ database table

Deficiency data is collected from the service outlets in each country and delivered as a monthly snapshot to the consolidated database. Unfortunately, due to a bad design of the data collection processes, information about when the deficiencies were detected or which records are actually new or changed is not available. The database was maintained since 1970, and it turned out that some deficiencies have been found as late as after 20 years. The database grew to about 200 Gigabytes nowadays. It is mirrored at some sites for evaluation purposes, but distributing the whole database after each new snapshot would be too expensive and online access was considered to be not secure enough. So the analyzer tool is used to detect the actually changed and new records. The generated updates, which are compressed and only about 500KB/day, are then distributed at a high security level.

The following code fragment shows how the compare mechanism is plugged together from the compare components to detect which records were actually changed. It is assumed that the consolidated data is in the database “main”, schema “topsecret” and new snapshots for Germany and the USA have been loaded into a separate database “snapshots”.

```
oldCon = new DBConnection( "jdbc://.../main", "topsecret" );
newCon = new DBConnection( "jdbc://.../snapshots", "topsecret" );

iterate = new DBCompareMethodIterating( new String[] {"company","factory","vehicle"} );
partition = new DBPartitionByColumnsPartial( new String[] {"country"}, iterate );
compareTable = new DBCompareTable( "deficiency", partition );

compareSchema = new DBCompareSchema( oldCon, newCon, new DBCompareTable[] { compareTable } );

compareSchema.getUpdates( theUpdateBuilder );
```

That example code also demonstrates how easy it can be to build a compare tool for yet unknown problem domains by only plugging together the database compare components described here.

Diagram 6.17 illustrates the interaction after “getUpdates()” is called.

In general, the interaction between compare components consists of a ’down chain’ of method calls where the ’compareRecords’ method of each partition is called until the final compare algorithm is reached, and two ’up chains’ of method calls through all the partitioning components when the

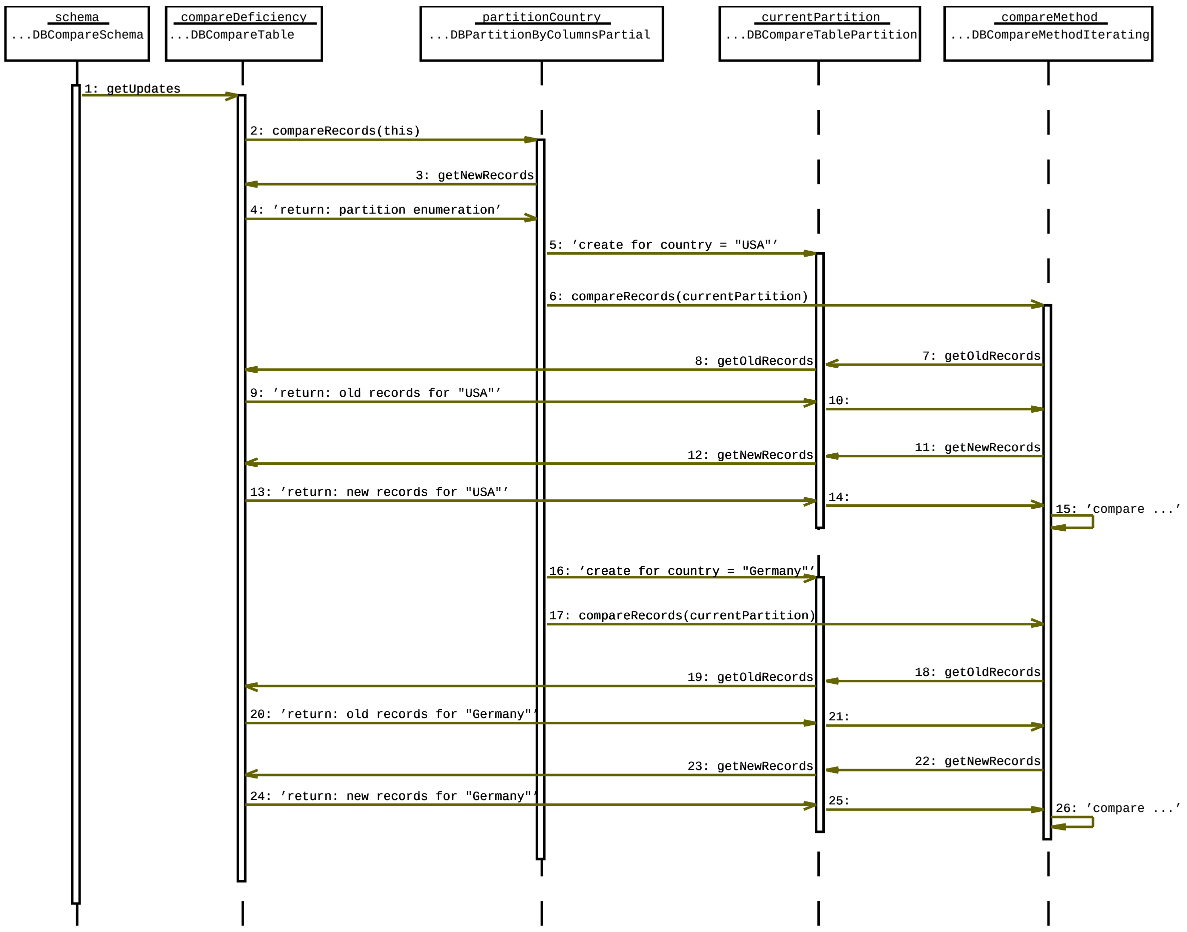
schema
...DBCompareSchema
compareDeficiency
...DBCompareTable
partitionCountry
...DBPartitionByColumnsPartial
currentPartition
...DBCompareTablePartition
compareMethod
...DBCompareMethodIterating
1: getUpdates
2: compareRecords(this)
3: getNewRecords
4: 'return: partition enumeration'
5: 'create for country = "USA"'
6: compareRecords(currentPartition)
7: getOldRecords
8: getOldRecords
9: 'return: old records for "USA"'
10:
11: getNewRecords
12: getNewRecords
13: 'return: new records for "USA"'
14:
15: 'compare ...'
16: 'create for country = "Germany"'
17: compareRecords(currentPartition)
18: getOldRecords
19: getOldRecords
20: 'return: old records for "Germany"'
21:
22: getNewRecords
23: getNewRecords
24: 'return: new records for "Germany"'
25:
26: 'compare ...'

Figure 6.17: Example for interaction between compare components

compare algorithm requests the records from the old and the new version of the table, respectively. In these up chains, each partition eventually adds its own partitioning constraints to the arguments of the get…Records call, using the compound DBAndConstraint described within the section about the database abstraction layer.

Descriptions of Individual Compare Components

The following are descriptions of the currently implemented database compare components:

DBPartitionByColumnsPartial compares only the partitions that are in the new version of the table. Some columns are given at construction time which identify the partitions. It is obviously not possible to detect deleted partitions using this partitioning scheme.

DBPartitionByColumns looks for partitions in the old and in the new version of the table. It calls the next compare (or partitioning) method in the chain for each partition that is either present in the old version, in the new version, or in both versions. Partitions are also identified by a set of columns given at construction time. The DBPairRecords class described in the next section is used to identify whether a partition was deleted, changed, or newly created.

DBCompareMethodPartial can only detect changed or newly created records. For each record found in the new version it searches for in the old one. Because a getOldRecords call is generated for each new or updated record, this method should only be used if the number of old records is much larger than the number of changed or new records. But even in this case trying to find well-sized partitions is the preferable way because it is much faster.

DBCompareMethodIterating iterates over the records found in the old and the new version and detects deleted, updated and inserted records. Records must be uniquely identified by the values found in the column(s) given at construction time. Otherwise, an exception is thrown. Record moves can also be detected if anything else than the primary key is used to identify the records. This class uses the utility class DBPairRecords to determine whether a record was deleted, changed, or newly inserted. A switch may be given when constructing instances of this class which will suppress generating record deletes. This enables supplying only a subset of the data as new data.

DBCompareMethodFindBestMatch is the most powerful compare method yet but the most difficult to use. As the name suggest, it tries to find the best match between the records from the old and the records from the new version. Records for which a counterpart could be found are considered updated and/or moved, the others are considered as deleted respectively inserted ones. Details about how this algorithm works can be found in the section 'Finding Best Matching Records' below.

The currently implemented compare components all sort, partition, and/or compare based on the values contained in one or more columns. If a different approach is needed, based on e.g. combined or derived values, the present classes can easily be extended to accommodate this by subclassing them or generating new implementations of the interfaces. Also, the design allows to use yet unknown compare algorithms specialized for the individual problem domains.

Finding Matching Record Pairs.

This section describes how the DBPairRecords utility works. This utility is used to find matching pairs of records or record groups. What is meant by a 'matching pair' is also explained in this section.

In general, there are too many records in the databases to be all held in memory. So the database compare components described above work on record streams (DBRecordEnumeration), which guarantees scalability, as the size of the databases is principally unlimited. They read a stream of records from the old and the new version, respectively.

Each of those partitioning or compare algorithms uses some discriminating columns to determine whether a pair of records respectively record groups found in the old and new version belongs together. The records read from the databases are first sorted (ascending) by these columns[3], then the values in these columns are compared. If they match between the old and the new version, a record pair was found. If they match among several records from the old and the new version respectively, a matching pair of record groups was found. But what if they do not match ? Which of the records should be considered as inserted or deleted ? The answer is easy: If the record read from the old version is smaller than the other, it was deleted, if the record from the new version is smaller, it was

[3]Note that it is important for these algorithms that the records from the old and the new version of the database are delivered using the same sort order. Databases from different vendors and/or different database codepages might produce incorrect results when string columns are used as sort keys. Also, the sorting of NULL values differs among DBMSs.

inserted. The following example demonstrates this, using a single numeric column for sorting and discriminating:

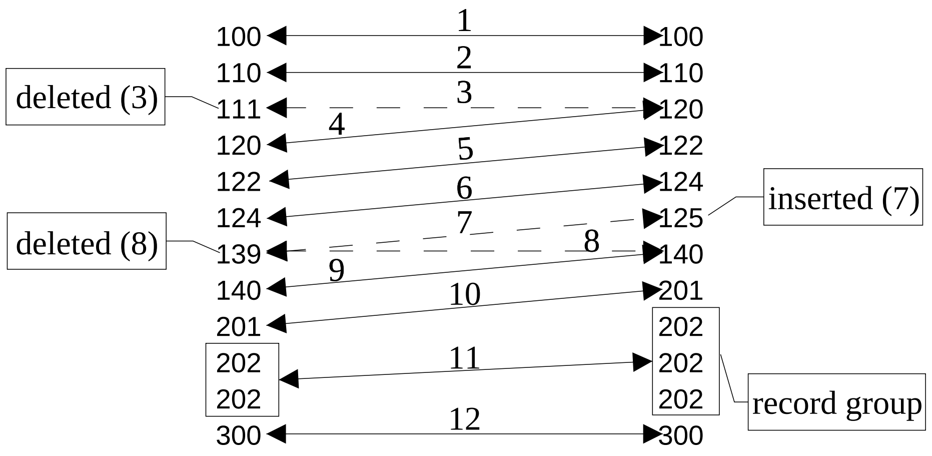

Figure 6.18: Example demonstrating algorithm for matching record pairs

This works nice for numeric columns, because every database sorts this type of column in the same way. But this is not the case for strings columns. The sort order of string columns depends on the codepage and collation order selected when the database was created. There is no way to determine via the JDBC driver how the database compares strings, and simulating all the available collations would be a time consuming and error-prone task anyway. Another possibility would be to ask the database to compare the strings, but this would be very slow. And even that wouldn't work. When I was searching for a way to compare strings that would be the least common denominator of the big database management systems, I dropped the idea of comparing strings immediately after I read the Oracle manual describing the difference between a VARCHAR and a VARCHAR2 column. However, how VARCHAR columns are sorted depends on the version (!) of the database server used:

> "The VARCHAR datatype is currently synonymous with the VARCHAR2 datatype. However, in a future version of Oracle, the VARCHAR datatype might be changed to use different comparison semantics."

I decided to use another approach. I implemented a lookahead algorithm in DBPairRecords which reads and stores records from both streams until it finds the smallest matching or comparable pair. For each record read from one of the streams, all records from the other stream stored already are tried, until a comparable pair is found. Note that equal records are always comparable. Then it is decided which records are deleted or inserted in a way similar to that described above.

Unfortunately, this algorithm has n^2 complexity, where n is only the number of inserted or deleted records, not the total number of records contained in the partition currently compared. If nothing changed between the versions, the complexity is just 1, because the first pair read is then already matching. If one of the partitions is empty, the complexity is 1 as well.

Finding Best Matching Records

The compare algorithm DBCompareMethodFindBestMatch tries to find the best matching records from the old and the new version of the data. It first sorts and groups the records by the columns defined at construction, using the DBPairRecords utility described in the last section. It then uses an arbitrary implementation of DBCompareBestMatchingRecords which finds the best matching records from the old and the new record group and compares them. Single records or whole record groups for which no match could be found are remembered for later processing. When the end of the partition is reached, all records for which no match could be found are given a second time to that implementation, since some records might have changed their corresponding record group between the old and the new version.

It can be a difficult thing to find the right attributes which produce small record groups for the first pass, but don't produce too much records which change their record group and must be processed in the second pass. A good start is to use attributes which change very seldom and describe a record as accurate as possible.

Class diagram 6.19 shows the currently implemented classes which are used to find the best matching records in a record group.

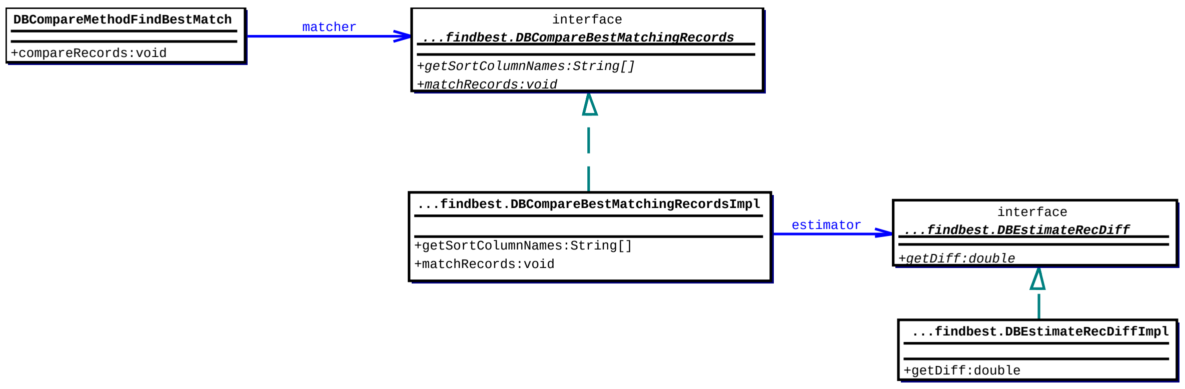
DBCompareMethodFindBestMatch
+compareRecords:void
matcher
interface
...findbest.DBCompareBestMatchingRecords
+getSortColumnNames:String[]
+matchRecords:void
...findbest.DBCompareBestMatchingRecordsImpl
+getSortColumnNames:String[]
+matchRecords:void
estimator
interface
...findbest.DBEstimateRecDiff
+getDiff:double
...findbest.DBEstimateRecDiffImpl
+getDiff:double

Figure 6.19: Classes used to find best matching records

The currently only implementation for finding the best matching pair of records in a pair of record groups is DBCompareBestMatchingRecordsImpl. It uses an instance of DBEstimateRecDiff to estimate the difference between an old and a new record. The getSortColumnNames method provides a hint for the sort order in a record group which helps completing the search for matching pairs fast. The following algorithm is used to find the best matching pair. It works on the two input arrays given to the matchRecords method described here.

1. For each pair processed, it first checks if the estimated difference is below a given limit. If so, the record pair is immediately compared and removed from the input.
   This guarantees a linear complexity if there was no change between the versions.

2. The estimated difference together with the indices of the record pair is inserted into a priority queue, if the difference doesn't exceed a defined limit. Step 1 and 2 are executed until the differences between all pairs have been calculated and all valid pairs have been put into the queue.

3. The pair with the lowest difference is selected from the queue.
   If one of the two records from that pair was already matched with another record, the pair is discarded. This is why the indices into the input arrays are put into the queue, and not the references to the records: otherwise, already matched records couldn't be detected.
   If both indices are valid, the pair is compared, which means, the changes between the two

records are generated as updates.
This continues until either the priority queue is empty, or there are no records left which could be matched.

This algorithm is based on a local optimization. Its worst case complexity is n^2, where n is the number of records in the input arrays. If one of the input arrays is empty, the complexity is 1, since there are no pairs that could be matched. The local optimization can produce far from optimal results when the differences calculated by the estimator are not adjusted wisely. The section 6.5.3 about comparing the footnote tables contained in the EPC database gives an idea of what might happen when inadequate differences are estimated.

The currently only implementation available for estimating the difference between two records is DBEstimateRecDiffImpl. It just adds on the differences calculated for each value contained in the records. By default, each value which is different between the two records is weighted with 1.0. But for some special columns specified at construction, change penalties (DBColumnPenalty) can be given which consist of a weight and a limit. The DBValue.estimateDiffTo() method is used for these columns to calculate the difference. This difference is then multiplied with the given weight but cut off at the given limit. As already mentioned above, the section about comparing the EPC database contains some examples for using this mechanism.

6.4.4 Building Database Updates

This section describes the database update component which is delivered to the update builder subsystem. Diagram 6.20 shows the involved classes.

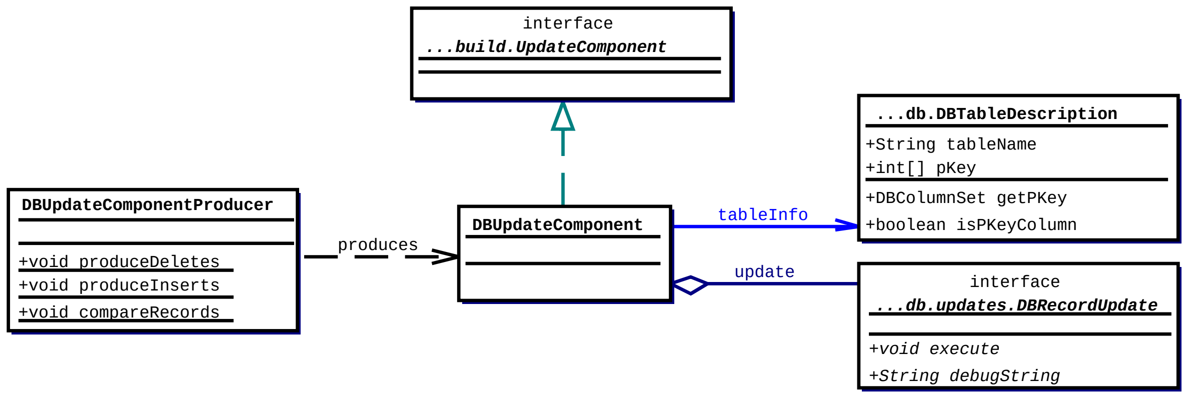

Figure 6.20: Class diagram showing database update component.

To be accepted by the update builder subsystem, the database update component has to implement the tagging interface UpdateComponent. The database updates are produced by the DBUpdateComponentProducer. Its methods shown on the diagram are called by the compare components described in the previous section, whenever deleted or inserted records were detected, or two records should be compared.

The update component contains a record update operation (DBRecordUpdate), which was described in the section about database updates. It also contains a reference to a description of the table the update operation belongs to. Otherwise, the update builder wouldn't know to which table the record update operation belongs. Putting the information about the table into DBRecordUpdate would be too expensive when the updates are serialized.

6.4.5 Implementations of the Temporary Update Queue Manager

Class diagram 6.21 shows the implementations of the TmpQueueManager interface.

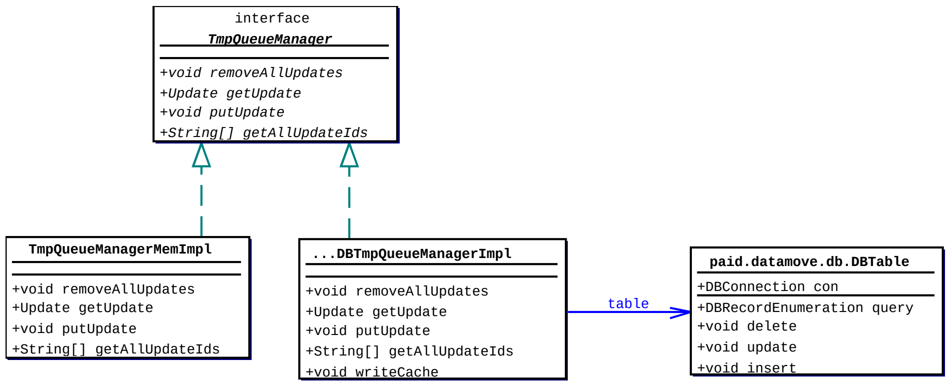

Figure 6.21: Implementations of TmpQueueManager.

The DBTmpQueueManagerImpl stores the updates into a database table. Since the updates must be serialized and deserialized to be stored into the database, this can be rather slow. A faster alternative is TmpQueueManagerMemImpl, which uses a hashtable to store the updates. But this implementation cannot always be used, since, depending on the problem domain, the generated updates can become too big to be held in memory.

An example for the use of the temporary queue manager can be found in section 6.2.1, describing the general update generator subsystem.

6.4.6 Implementation of the Public Update Queue Manager

Class diagram 6.22 shows the currently only implementation DBUpdateQueueManagerImpl of the UpdateQueueManager interface.

That implementation uses a database table to store the updates permanently for public access. The performance problem is the same as mentioned for the temporary update queue above.

The UpdateQueueFilter interface was introduced to allow a preselection of updates before they are retrieved from the database. The updates can be preselected based on the information that is passed to the accept method, which is the update identifier, the timestamp, and the information about newly introduced data subsets. The local data description LocalDataDesc implements this interface and can be used directly to select the appropriate updates from the queue. The sequence diagram shown for the SimpleApply subsystem (6.4) illustrates this.

6.4.7 LocalData Implemented for Relational Databases

DBLocalData is used to represent local data that is stored within a relational database. It is used by the Smart Updates to get a reference to the local data that should be updated. Class diagram 6.23 shows the involved classes.

The database represented by a DBLocalData object is accessed via an associated DBConnection object. The transaction supporting methods beginTransaction(), commit(), and rollback() are implemented by calling the respective methods on the DBConnection object. The description of the data stored in the database, a LocalDataDesc object, is stored permanently in a table contained in the same

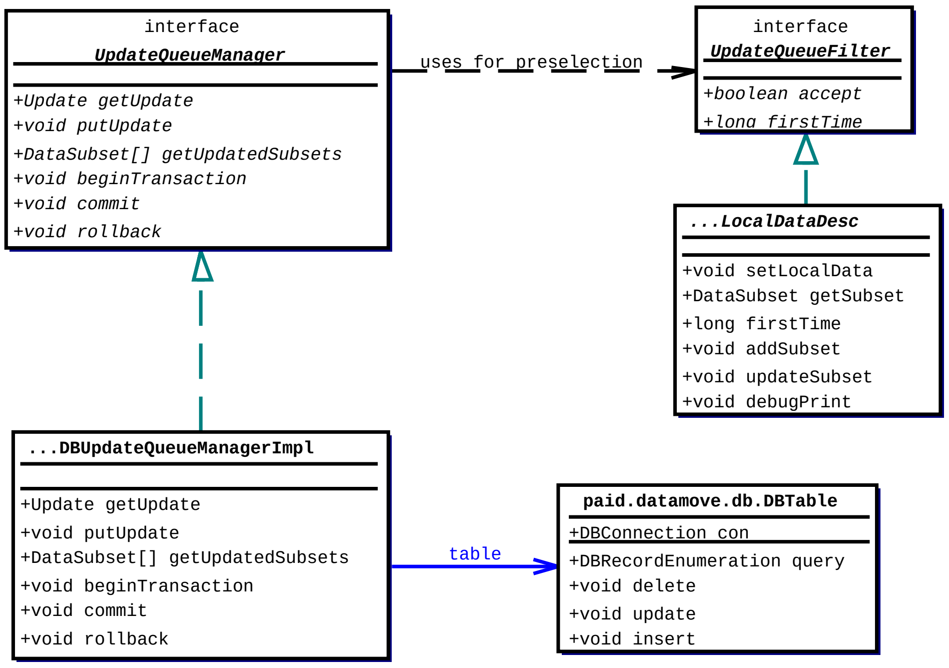

Figure 6.22: Implementation of UpdateQueueManager.

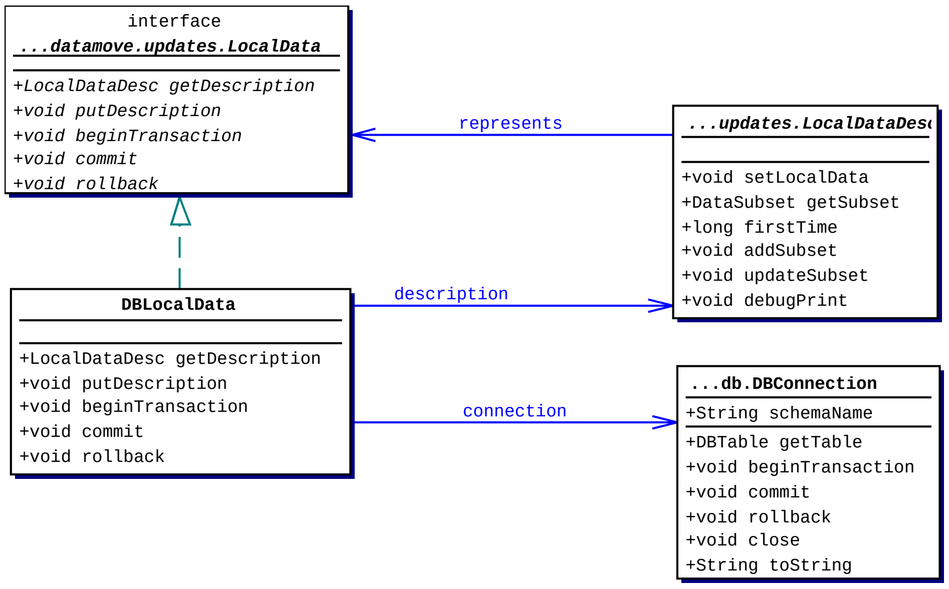

Figure 6.23: Classes associated with DBLocalData.

database. The getDescription() and putDescription() methods are used respectively to read or write that description to the database.

6.5 Problem Domain Specific Classes for EPC

This section describes the EPC specific implementations. These are the EPC Smart Updates, the main EPCAnalyzer class, and the EPC specific compare, build, and publish subsystems.

6.5.1 EPC Updates

The Updates to the StarParts database are structured like the subsets. The class diagram 6.24 shows their hierarchy.

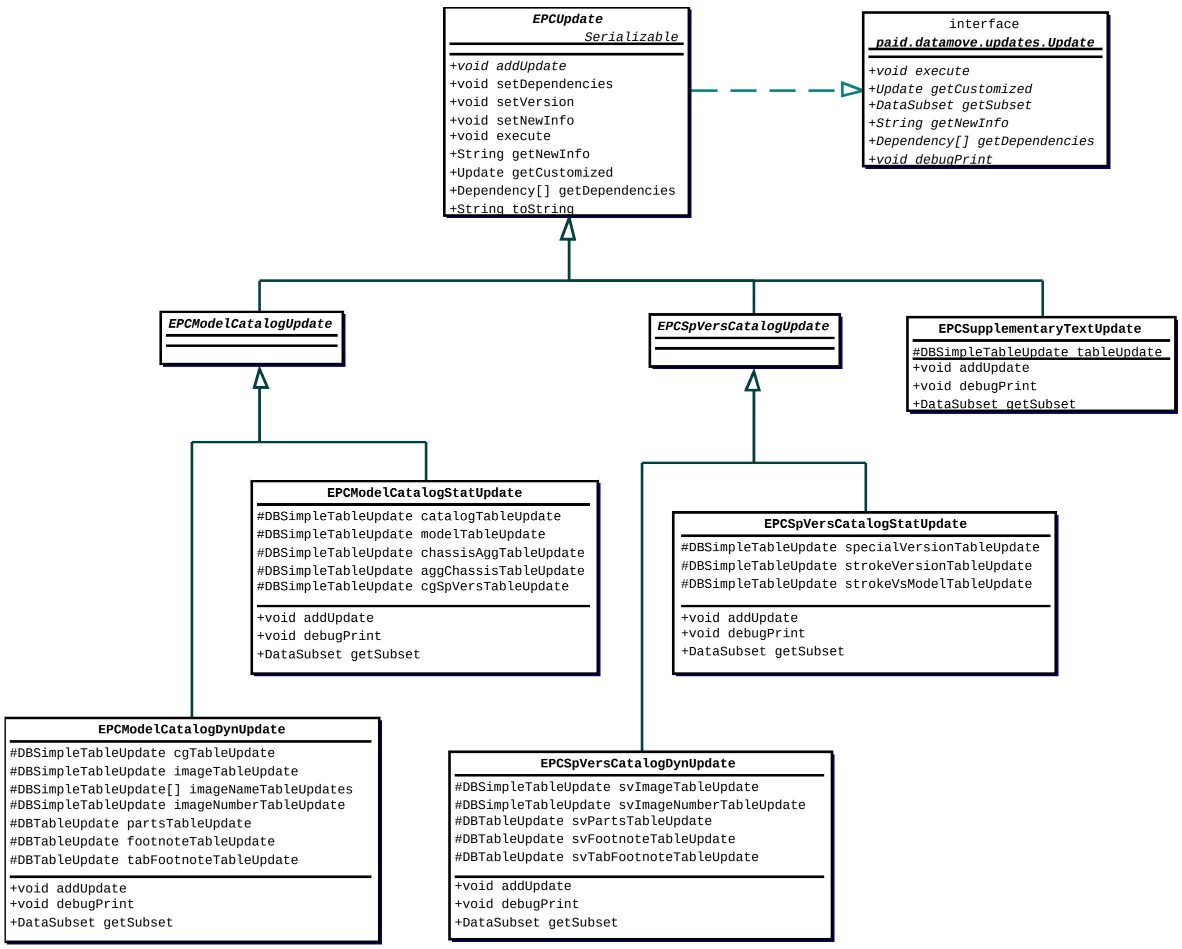
EPCUpdate
Serializable
+void addUpdate
+void setDependencies
+void setVersion
+void setNewInfo
+void execute
+String getNewInfo
+Update getCustomized
+Dependency[] getDependencies
+String toString
interface
paid.datamove.updates.Update
+void execute
+Update getCustomized
+DataSubset getSubset
+String getNewInfo
+Dependency[] getDependencies
+void debugPrint
EPCModelCatalogUpdate
EPCSpVersCatalogUpdate
EPCSupplementaryTextUpdate
#DBSimpleTableUpdate tableUpdate
+void addUpdate
+void debugPrint
+DataSubset getSubset
EPCModelCatalogStatUpdate
#DBSimpleTableUpdate catalogTableUpdate
#DBSimpleTableUpdate modelTableUpdate
#DBSimpleTableUpdate chassisAggTableUpdate
#DBSimpleTableUpdate aggChassisTableUpdate
#DBSimpleTableUpdate cgSpVersTableUpdate
+void addUpdate
+void debugPrint
+DataSubset getSubset
EPCSpVersCatalogStatUpdate
#DBSimpleTableUpdate specialVersionTableUpdate
#DBSimpleTableUpdate strokeVersionTableUpdate
#DBSimpleTableUpdate strokeVsModelTableUpdate
+void addUpdate
+void debugPrint
+DataSubset getSubset
EPCModelCatalogDynUpdate
#DBSimpleTableUpdate cgTableUpdate
#DBSimpleTableUpdate imageTableUpdate
#DBSimpleTableUpdate[] imageNameTableUpdates
#DBSimpleTableUpdate imageNumberTableUpdate
#DBTableUpdate partsTableUpdate
#DBTableUpdate footnoteTableUpdate
#DBTableUpdate tabFootnoteTableUpdate
+void addUpdate
+void debugPrint
+DataSubset getSubset
EPCSpVersCatalogDynUpdate
#DBSimpleTableUpdate svImageTableUpdate
#DBSimpleTableUpdate svImageNumberTableUpdate
#DBTableUpdate svPartsTableUpdate
#DBTableUpdate svFootnoteTableUpdate
#DBTableUpdate svTabFootnoteTableUpdate
+void addUpdate
+void debugPrint
+DataSubset getSubset

Figure 6.24: Smart Updates for EPC database

The updates serve mainly as containers for DBSimpleTableUpdate and DBTableUpdate objects, which in turn store the record updates to the EPC database tables. The only interesting function is the getCustomized() method. It creates a visitor which visits the updates to tables which contain language specific data. This visitor then removes the updates to language specific columns or records for languages that are not selected by EPCLocalDataDesc.languages.

The LocalData object passed to the execute() method must be an instance of EPCLocalData, which is a subclass of DBLocalData which was introduced only to allow to check whether the right LocalData reference was passed to the update.

Dependencies and Information About New Subsets

The dependencies and the information about newly introduced subsets contained in an EPCUpdate is initialized with setDependencies() respectively with setNewInfo() by the publisher subsystem (EPCPublisher, section 6.5.5). The latter is currently not implemented; updates will only return the string ’NEW’ if they introduce a new subset.

Each EPCUpdate which doesn’t introduce a new subset depends on the previous version of the subset it will update. Two updates to the static and dynamic subsets of the same catalog depend on one another, and each catalog update depends on the current version of the supplementary texts table.

6.5.2 Toplevel Class EPCAnalyzer

Class diagram 6.25 shows the top level class EPCAnalyzer and the associated classes of the different subsystems.

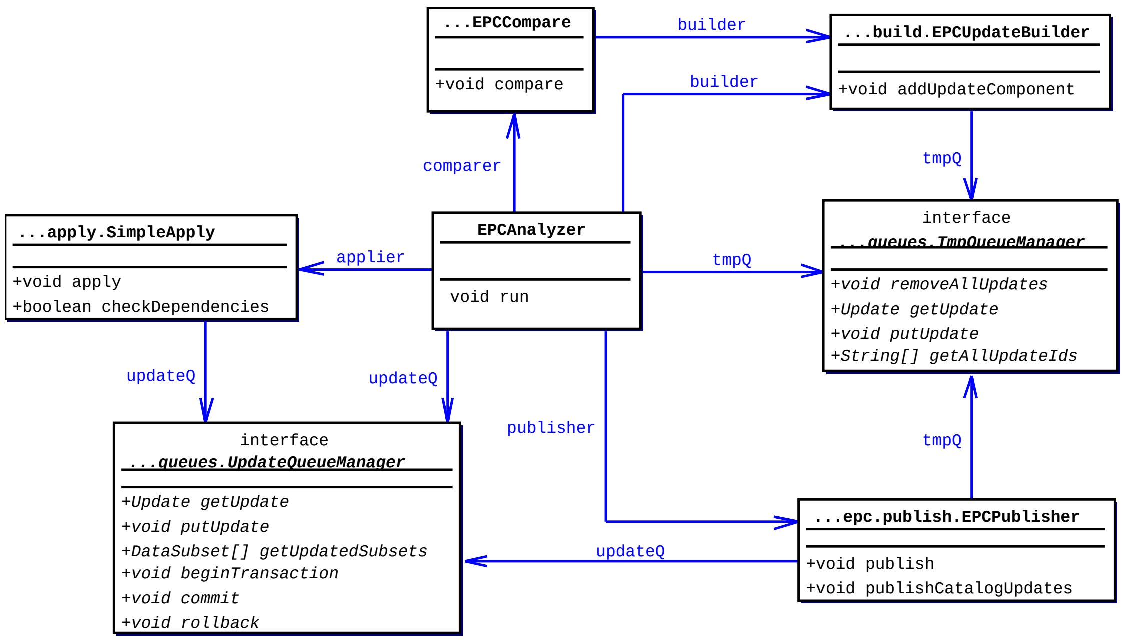

Figure 6.25: Main EPC analyzer class and associated subsystems.

The very simple sequence diagram 6.26 shows how the activity diagram 6.2 found in the section about the subsystem decomposition maps to method calls on the subsystem classes. The activities were mapped directly to method calls.

6.5.3 Compare Subsystem

The main task of the EPC specific implementation of the compare subsystem is to specify the parameters for the database compare components. This section is split up into four subsections: comparison of part record tables, footnote tables, the supplementary texts table, and the remaining (’standard’) tables.

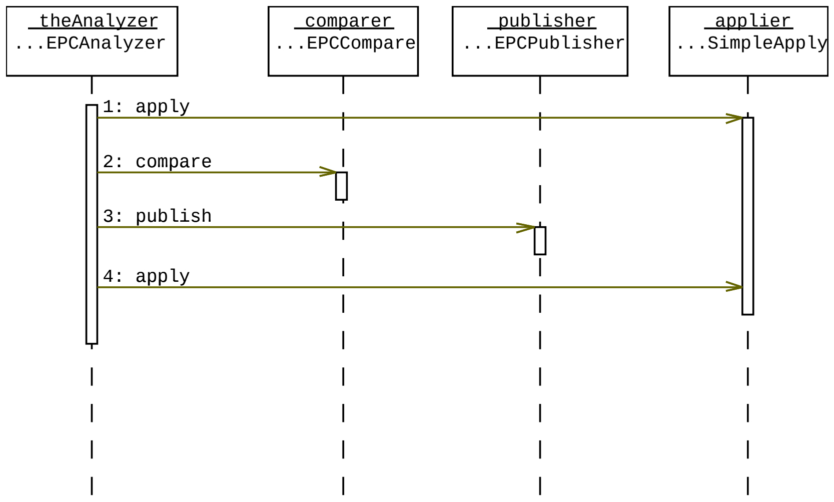

Figure 6.26: Sequence of top-level method calls to analyzer subsystems.

Standard Tables Compared by Primary Key

All tables which are not a part record, footnote, or supplementary texts table are compared by iterating over the primary key in the old and the new version of the table, respectively. Where applicable, tables are subdivided partially by catalog and construction group. This enables using an incremental construction group supply, as the new versions of the table can contain only data for changed construction groups. The tables for model specific data are only subdivided partially by the catalog and the special version tables by the special version main number, because the supply process always supplies whole special version catalogues as one unit.

Supplementary Texts Table

This table is compared using the DBCompareMethodPartial database compare component, which selects for each record found in the new table the appropriate record from the old one. It therefore can generate only updates and inserts. Also, as already mentioned, it is very slow. The reason for choosing this method was, that some catalogues contain supplementary texts inline. These are extracted and put into the new version of this table. Because that are only very few, using this method was appropriate. When a new version of the supplementary texts is supplied, the comparison may take some time, but that happens only every few month. Also, if the date fields in the supplementary texts table were supplied correctly, one could delete all records which are too old before running the analyzer.

Part Records

The problem with part records is that they have no unique identity. Instead, a running number is used to identify records within a construction group. Today, many of these running numbers change when image groups are reorganized (see chapter 5 for details). For the planned conversion from DIALOG to the ELDAS format, these running numbers will be given only temporarily for each converted snapshot. Given that, I implemented a mechanism which searches for the part records, DBCompareBestMatchingRecordsImpl. A detailed description of the algorithm can be found in section 6.4.3.

Principally, one could use this algorithm directly for each construction group to find the best matching records. But, as this algorithm has n^2 complexity, this could get very slow for large construction groups, since they may contain more than 2,000 part records. So I use the following attributes to group the part records into smaller units: the part number and the flags indicating whether a part is valid for manual or automatic transmission, and whether it is valid for left hand or right hand steering. I chose this attributes because they change very seldom. Grouping the records by this values, I get the values for 'n' shown in table 6.2, where n is the size of one group.

n	1	2	3-5	6-10	11-20	21-40	41-80	81-191
occurrences	269,218	24,681	10,094	2,153	734	223	91	53
in percent	87.6	8.0	3.3	0.7	0.2	0.07	0.03	0.02

Table 6.2: Sizes of part record groups with number of occurrences.

The table shows how often 'n' occurs in our reference installation. 99% of all part records are grouped to less than 6 part records. 99,9% are grouped to less than 21 records.

Given that n is mostly 1 or 2, the n^2 complexity doesn't hurt. The problem in using more attributes to group the part records to smaller groups is that at the end of the construction group the algorithm has to match yet unmatched records again. This affects records which are really new or deleted, but also records where those attributes selected for grouping changed, since they will then be found in different groups in the old and the new version. So one has to find a good compromise to select the appropriate attributes. For the part records, I chose attributes that almost never change and specify a part records relative precisely. It is important that the underlying algorithm has only linear complexity when nothing was changed between two versions of a catalog, because this is the most common case.

This method is based on a local optimization, e.g. the algorithm tries to first find the best matching pair in a record group, then the second best, and so on. Using a global optimization to find the best match between two groups by considering all possible pairs has exponential complexity, which is not feasible.

Note that the semantics of record changes are much better represented by the updates generated by this method than by using the primary key to find matching records. But, as heuristics are used, this is not perfect. Actually, it can never be, since I discovered that sometimes even a human being cannot determine what really happened between two versions by looking at the resulting snapshots.

Footnotes

The problem with footnotes is that running numbers are used to identify and order the lines. If a line is inserted or deleted, all line numbers after that change will be different between the two snapshots.

The footnote tables are compared using the same heuristic algorithm as for part records (DBCompareBestMatchingRecordsImpl). The footnote groups are determined by the footnote number and the language, as these two attributes almost never change. Within each group, the algorithm then tries to find the best match between footnote lines.

It was very important to manually adjust the penalties evaluated by this algorithm. The standard weighting takes every column with 1.0 into account. Since there are 2 line numbers for the normal footnotes and 3 for the tabular footnotes, the difference between line numbers would weight stronger than that between the contents of the footnote lines. So the penalty for the latter was adjusted to be at least as big as the sum of the penalties for all line numbers.

The provided EPC schema had to be modified. The tables FUNO_TAB and SA_FUNO_TAB containing the table footnotes used a separate, language independent line number (BLOCK_ZEILE) as primary key. I changed the schema in a way that these tables now use the same columns as the normal footnote tables in addition to the grouping column BLOCK_ZAEHLER. Using the previous, language independent key, it would be impossible for the Smart Updates to determine the language of the generated DBUpdateSingleRecord update operations. This is because only updated and primary key columns are contained in such an update operation.

6.5.4 Update Builder Subsystem

The update builder subsystem consists of only one class: EPCUpdateBuilder. It just performs the following tasks:

- It receives a record update operation embedded in a DBUpdateComponent from the compare subsystem via the UpdateBuilder interface,
- It determines whether the record update is to a model catalog, special version catalog, or to the supplementary texts table. This is done with help from EPCSchema, which contains the information about which table belongs to which subset.
- For model catalog updates, the catalog identifier is determined, for special version catalog updates, the special version main number is determined.
- It tries to load the respective EPC Smart Update from the temporary queue. When none is found, a new one is created.
- It passes the record update to the Smart Update.
- And finally, the Smart Update is written back to the temporary queue.

6.5.5 Publisher Subsystem

This is also only one class: EPCPublisher. It's main task is to calculate the dependencies between the updates contained in the temporary queue, and to put them into the public update queue. The latter is done within one transaction using the beginTransaction() and commit() methods from the UpdateQueueManager for the public update queue.

It first gets all update identifiers from the temporary queue and uses a hashtable to find the matching updates for the static and dynamic subsets of the same catalog. An associated EPCLocalDataDesc object which describes the data contained in the reference database is used to determine which Smart Updates introduce new subsets, and to initialize the dependencies to existing subsets.

6.5.6 Helper Classes

The following helper classes were created to encapsulate EPC specific information at a defined place:

EPCLanguage holds all data which identifies languages, namely the prefixes and codes described in section 5.1.

EPCTableInfo describes a table in the EPC database. It contains information about which table belongs to which catalog type (model or special version) and if it belongs to the dynamic or static subset of a catalog.

EPCSchema is a container for all the table information objects. It also provides a fast mapping method getTableInfo() for getting the table info object for a given table name.

EPCColumns contains some of the column names found in the EPC database. It was introduced to avoid putting the database column names directly into the code.

6.6 Problem Domain Specific Classes for FDOK

This section describes the FDOK specific implementations. These are the FDOK Smart Updates, the main FDOKAnalyzer class, and the FDOK specific compare, build, and publish subsystems.

6.6.1 FDOK Updates

Class diagram 6.27 shows the updates to the StarIdent (FDOK) database. These updates serve mainly as a container for the DBSimpleTableUpdate objects, which in turn store the record updates to the FDOK database tables.

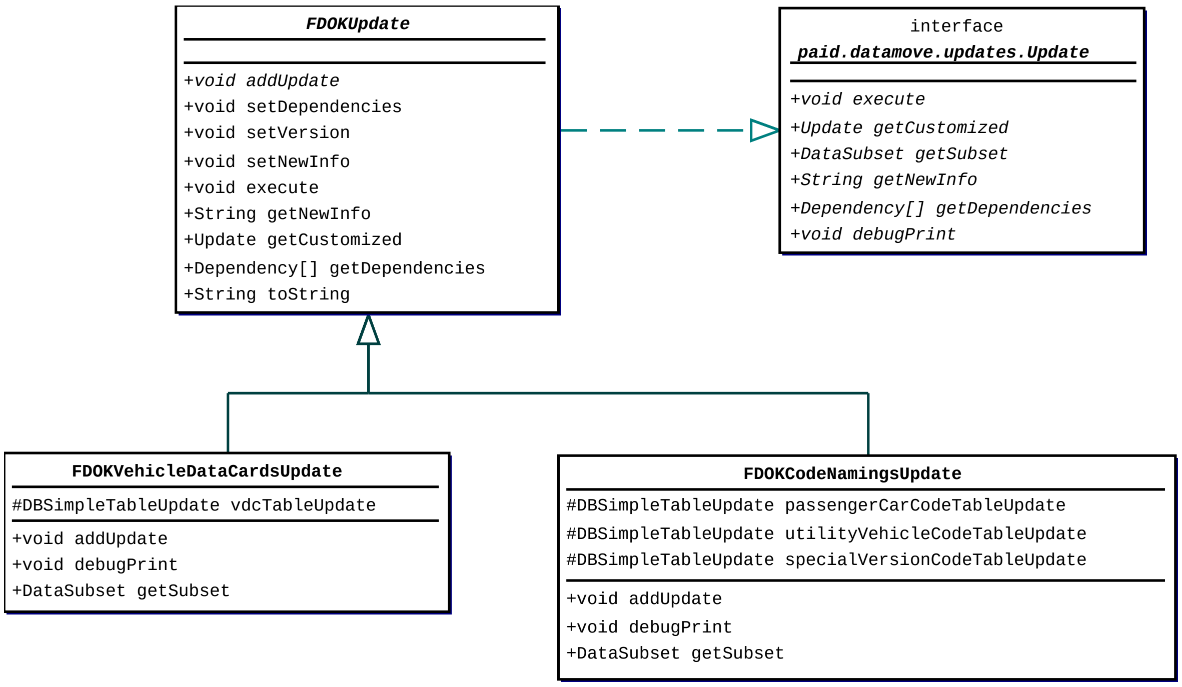

Figure 6.27: Smart Updates for FDOK database

The FDOKVehicleDataCardsUpdate is customizable both for single vehicle data cards (FDOK-SingleVehicleDataCard) and vehicle data cards subsets (FDOKVehicleDataCardsSubset). For the latter, the getCustomized() method does nothing (returns the same FDOKUpdate), for the former, it strips off all vehicle data card record updates from the Smart Update that are not locally installed. This prevents the update from installing unwanted vehicle data cards.

The FDOKCodeNamingsUpdate just updates the code naming tables for the passenger car, utility vehicle, and special version codes. As already mentioned in section 5.3.6 about the FDOK subsets, these should always be locally installed, because these tables are relatively small, only a few MB.

The LocalData object passed to the execute() method must be an instance of FDOKLocalData, which is a subclass of DBLocalData which was introduced to allow checking whether the right LocalData reference was passed to the update.

Dependencies and Information About New Subsets

The dependencies and the information about newly introduced subsets contained in the FDOKUpdate is initialized with setDependencies() respectively with setNewInfo() by the publisher subsystem (FDOKPublisher, section 6.6.5). The string returned by getNewInfo() is the identity of the updated subset (getSubset().getIdentity()).

Each FDOKUpdate which doesn't introduce a new subset depends on the previous version of the subset it will update.

6.6.2 Toplevel Class FDOKAnalyzer

The toplevel of the FDOKAnalyzer looks exactly the same as that for the EPCAnalyzer show in section 6.5.2, with the only difference that all occurrences of 'EPC' are replaced by 'FDOK'.

6.6.3 Compare Subsystem

The classes representing the compare subsystem for FDOK are FDOKCompare and FDOKSubdivideVDCPartial, located in the paid.datamove.fdok.compare package. The compare() method of the former is called by the main FDOKAnalyzer class to initialize the database compare components and start comparing the tables.

Comparing the Main Vehicle Data Cards Table

The FDOKSubdivideVDCPartial is used to subdivide the main vehicle data cards table before comparing it. It is an implementation of the DBCompareRecords interface and works very much like DBPartitionByColumnsPartial (see section 6.4.3), except that it doesn't use directly the values from some given columns, but substrings of those for determining the partitions. The same values (model line and area code[4]) which are used to specify FDOK vehicle data card subsets are used here to partition the table before comparing it. Only the partitions contained in the new version of the table are compared, so the new version may contain only some subsets of the vehicle data cards, namely those contained in the incremental updates coming from the legacy FDOK database.

Each of these partitions is compared by iterating over the primary key using DBCompareMethodIterating. The 'partial' flag is given to the latter to allow each partition in the new version to contain only a subset of vehicle data cards. Otherwise, delete record commands would be generated for the 'missing' vehicle data cards.

The DBCompareTable class was subclassed to CompareVDCTable, an inner class of FDOKCompare, to implement the following two extensions:

- The getPKey() method of DBCompareTable was overridden to pretend that the order number (AUFTRAG) is part of the primary key, which was actually only the vehicle identification number (FIN). This was done to allow the update builder subsystem (FDOKUpdateBuilder) to

[4]The build year, which is planned to be used for subsetting FDOK, is currently not available.

determine the area code from the order number. Otherwise, if that field was not changed between two versions of a vehicle data card, it wouldn't be included in a record update operation (DBUpdateSingleRecord) generated by the database compare components.

- As explained in the section about the size of the FDOK updates (5.3.4), a vehicle data card should be uncompressed before it is again compressed as part of an compound FDOK Smart Update.
  The standard DBAutoColumn handler assigned automatically to the column containing the gzipped Java objects would just store the compressed binary images into DBBinaryValue objects. So the getColumns() method of DBCompareTable was overridden to establish FDOKGzipColumn for this column, which uncompresses the values and produces FDOKGzipValue objects, a subclass of DBBinaryValue (see class diagram 6.28). The latter then contains the uncompressed binary image, and automatically compresses it again when writing to the database by overriding the fillPrepStmt() method. This mechanism is very similar to that described for DBObjectColumn and DBObjectValue in section 6.4.1.

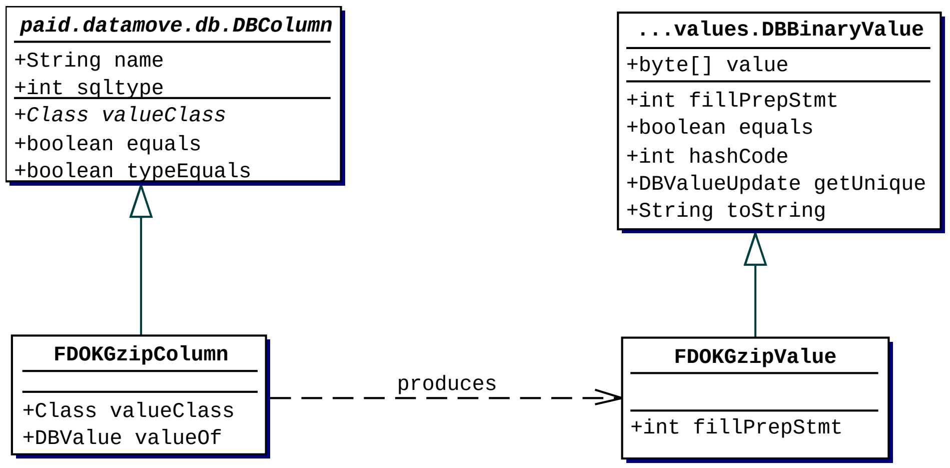

Figure 6.28: Classes used for handling gzipped vehicle data card objects.

Comparing the Code Naming Tables

The code naming tables are just compared the standard way by iterating over the primary key, using DBCompareMethodIterating.

6.6.4 Update Builder Subsystem

The update builder subsystems for FDOK consists of only one class: FDOKUpdateBuilder, located in the paid.datamove.fdok.build package. It implements the UpdateBuilder interface to receive the UpdateComponents from the compare subsystem. It performs the following steps to combine the UpdateComponents to FDOKUpdates:

- It receives a record update embedded in an UpdateComponent from the compare subsystem.
- It determines if the record update is an update to the vehicle data cards table or to one of the code naming tables. The information from the helper class FDOKSchema is used for this purpose.

- For vehicle data card updates, the model line and area code is determined from the primary key, which was prepared by FDOKCompare (see above). This information is needed to assign the record update to the correct FDOKVehicleDataCardsSubset.
- Like shown in the general interaction diagram for the update generator subsystem (6.3), it first tries to load the appropriate FDOKUpdate from the temporary queue. If none was found, a new FDOKUpdate is generated.
- The record update is passed to the FDOKUpdate via addUpdate().
- The Smart Update is written (back) to the temporary queue.

6.6.5 Publisher Subsystem

The publisher subsystem is implemented by the FDOKPublisher class in the paid.datamove.fdok.publish package. After the FDOKUpdate objects are generated by the compare and update builder subsystems, this subsystem is activated. It's main task is to calculate the dependencies between the updates contained in the temporary queue, and to put them into the public update queue. The latter is done within one transaction using the beginTransaction() and commit() methods from the UpdateQueueManager for the public update queue.

6.6.6 Helper Classes

The following are only helper classes to encapsulate meta information about the FDOK database (e.g. schema, table, and column names):

FDOKTableInfo describes a table in the FDOK database. It tells whether a table is a code naming table or the vehicle data card table.

FDOKSchema is a container for the table information objects. The getTableInfo() method can be used to determine the FDOKTableInfo for a given table name.

FDOKColumns contains some of the column names from the FDOK database. It was introduced to avoid putting the database field names directly into the code.
It also contains the offsets into the vehicle identification number (FIN) and order number (AUFTRAG) strings to calculate the model line respectively the area code from these attributes.

Chapter 7

Analyzer Usage and Sample Data

This thesis is accompanied with either a CD-ROM or with archive files containing the following items:

- The analyzer source code and compiled '.class' files.
- The javadoc documentation.
- Shell scripts and SQL scripts.
- The modified EPC/FDOK conversion utilities and scripts.
- Sample data for EPC and FDOK.
- The original conversion utilities and scripts supplied by debis.

The next sections will explain where these items are located, and which sample data and utility scripts were provided.

7.1 Directory Hierarchy

The following list of directories is relative to the root directory 'analyzer/' found on the CD-ROM or in the archive files, respectively.

src/ Contains the source code directory hierarchy.

classes/ The compiled Java '.class' files are located here.

doc/ This is the root directory of the javadoc documentation.

scripts/ Contains the main shell scripts, which are described in the sections below.

config/ Contains some configuration scripts sourced by the shell scripts. These configuration scripts must be customized to reflect the local database configuration.

sql/ All SQL scripts for creating and importing into databases are located here.

epcconv/ Conversion 'awk' scripts for EPC, as described in 5.2.8.

epcdata/ Sample EPC catalogues.

fdokconv/ Modified Java classes used to load the database with the incremental updates coming from the legacy FDOK database (described in 5.3.2).

fdokdata/ Sample data for FDOK.

orig/ Some of the original data and scripts supplied by Mercedes and debis.

7.2 Configuration

The scripts were created and customized for the ’MySQL’ database, assuming that a user ’analyzer’ with the same password has access to the databases used for the data and queues.

For using these utilities with other database systems, the scripts located in the ’config/’ directory must be customized. The comments contained in these scripts should help doing this. Depending on the database system used, other shell scripts and SQL scripts might have to be customized as well.

7.3 Sample Data and Utilites for EPC

7.3.1 Sample Catalogues

The sample model and special version catalogues are located in directories below ’epcconv/init/’ and ’epcconv/updates/’. The former holds all catalogues which should be loaded initially into a reference database, using the ’EPCImport’ script described below. The latter contains the update snapshots to these catalogues which can then be compared to the reference database using the ’EPCImportAnalyze’ script.

These catalogues are stored in directories reflecting the time when they were extracted from the ELDAS system. These are examples for initial data and updates of special version catalogues and a model catalog (’30L’):

```
epcdata/init/19980305.090725/SA.TXT.gz
epcdata/updates/19980313.084923/SA.TXT.gz
epcdata/updates/19980320.075553/SA.TXT.gz
epcdata/init/19980423.180730/30L.TXT.gz
epcdata/updates/19980428.130109/30L.TXT.gz
epcdata/updates/19980504.163123/30L.TXT.gz
epcdata/updates/19980728.094821/30L.TXT.gz
epcdata/updates/19980728.094822/30L.TXT.gz
```

7.3.2 Utilities

The following EPC specific scripts can be found in the ’scripts/’ subdirectory:

EPCCreate

This script creates the EPC tables within a new database.

Parameters: <database name>

EPCImport

Imports one file containing a model catalog or special version catalogues into an EPC database.
Parameters: <filename> <database name>

<filename> may be for example 'epcdata/init/19980423.180730/30L.TXT.gz'.

EPCCreateDataDesc

This script creates the table which contains the description of the data (EPCLocalDataDesc) contained in the database. It then invokes the Java utility paid.datamove.epc.run.EPCCreateDataDesc, which detects which catalogues are present in the database and creates a new EPCLocalDataDesc object within that table.
Parameters: <database name>

CreateQueues

Create the tables which will contain the temporary queue and the public update queue.
Parameters: <database name>

EPCImportAnalyze

This is the main script, as it runs the EPCAnalyzer tool. It creates the EPC tables within a temporary database and imports all catalogues contained in the given directory into that database. It then runs the analyzer tool paid.datamove.epc.run.EPCAnalyzer, which will compare that database against the given reference database, generate the EPCUpdates reflecting the differences between them, put them into the given update queue, and finally apply them to the reference database.
Parameters: <catalog directory> <reference database> <temp. database> <update queue database>

<catalog directory> may be e.g. 'epcdata/updates/19980428.130109'.

7.4 Sample Data and Utilities for FDOK

7.4.1 Sample Data

The following sample data can be found in the 'fdokdata/' subdirectory:

areacodes: list of the areacodes that can be found in the order number attribute of a vehicle data card.

star.pkw.code.gz: data for the distribution code naming tables for passenger cars.

star.nfz.code.gz: data for the distribution code naming tables for utility vehicles.

star.saben.gz: data for the special version code naming table.

pkw.gz: some vehicle data cards for passenger cars updated between 11/04/97 and 11/14/97.

star.pkw.data.gz: vehicle data cards for passenger cars updated between 14/09/98 and 10/20/98.

star.nfz.data.gz: vehicle data cards for utility vehicles updated between 14/09/98 and 10/20/98.

7.4.2 Utilities

The following FDOK specific scripts can be found in the ’scripts/’ subdirectory:

FDOKCreate

This script creates the FDOK tables within a new database.
Parameters: <database name>

FDOKImport

Imports one file containing updated vehicle data cards into a FDOK database.
Parameters: <filename> <database name>

<filename> may be for example ’fdokdata/star.pkw.data.gz‘.

FDOKCreateDataDesc

This script creates the table which contains the description of the data (FDOKLocalDataDesc) contained in the database. It then invokes the Java utility paid.datamove.fdok.run.FDOKCreateDataDesc, which detects which vehicle data card subsets (FDOKVehicleDataCardsSubset) are present in the database and creates a new FDOKLocalDataDesc object within that table.
Parameters: <database name>

CreateQueues

Create the tables which will contain the temporary queue and the public update queue (was already mentioned above, for EPC).
Parameters: <database name>

FDOKAnalyze

This is the main script, as it runs the FDOKAnalyzer tool, paid.datamove.fdok.run.FDOKAnalyzer. It will compare a temporary database containing updated vehicle data cards to the given reference database, generate the FDOKUpdates reflecting the differences between them, put them into the given update queue, and finally apply the updates to the reference database.
Parameters: <time> <reference database> <temp. database> <update queue database>

<time> is the version timestamp for the generated updates.

Chapter 8

Conclusion

The problem of the aftersales data snapshots that were too big for world-wide distribution was successfully solved by the analyzer tool developed within this thesis. The various possible ways of customizing and extending the analyzer algorithms and the generated updates makes it possible to use this tool for other types of aftersales data as well.

Using the analyzer tool is most interesting in problem domains where the location of data is changed between different versions of the database. It may not only be used for generating updates, but also for answering the question, what really changed between two versions of the same data. This question cannot always be answered easily in these problem domains without such a tool.

An example for such a problem domain is the service information database ’WIS’. Here the problem of matching the data between different versions is worse than for the Electronic Parts Catalogues. This may be an interesting application domain for the analyzer tool.

The updates generated by the analyzer tool use Java as implementation language and JDBC to access the databases. This is a platform and vendor independent approach, which enables to replicate updates to all kinds of databases. It may be possible to extend this approach to automatically generate updates from transactions made to a source database and thus implement a truly vendor neutral replication mechanism. There are at least two ways to do this: One could either generate the updates from the transaction log of the database, or provide an intermediate JDBC driver which replicates the update statements made to a source database.

Chapter 9

Related Work

The major areas of interest in the context of this thesis were change detection, disconnected operation, synchronization, and replication techniques.

Local Change Detection

Several tools and techniques are available for local change detection. The approach is similar to that used in this thesis. The differences between two versions of data are generated locally and can then be distributed to those who have the old version already, and want to get the new one without downloading the whole new data.

- There is the well-known Unix ’diff’ command, which detects and generates the differences between two text files.
- Some tools are available for detecting the differences between binary files as well, for example xdelta [McDo], which uses a block checksum algorithm based on rsync [TM96].
- In [CG97], techniques are presented to do meaningful change detection on hierarchical structured data, e.g. a document - paragraph - sentence hierarchy.
- There are also comparison techniques available for relational databases. Transaction Software for example offers a simple ’dbdiff’ tool for their TransbaseCD database [Tra98], which generates the differences between two database versions as a SQL script which can then be distributed and applied to remote databases. This tool is not usable for the Mercedes-Benz aftersales databases, because it assumes that the primary key does not change between releases.

Disconnected Operations in Distributed Filesystems

The CODA filesystem [Kis96] was developed to support disconnected operation. Changes made on a client while not connected to the server are gathered and later replayed when contacting the server again.

Other approaches are the Ficus distributed filesystem [GHM+90], and it’s descendant, Rumor [RPG+96], which does user-level file and directory synchronization based on a reconciliation mechanism.

Synchronization

There are tools and techniques available to allow synchronization of files and databases.

- Rsync [TM96] uses a fast block checksum algorithm to efficiently synchronize files (and directories) over a network.
- The snc tool [BP98] can be used in mobile computing environments to synchronize filesystems after a disconnected operation. It uses reconciliation like the Rumor filesystem. It is available as a Java tool with a few native methods.
- The [BR97] paper presents several techniques to perform the comparison of two database copies that are connected over a relatively slow WAN network link. The techniques use checksums of records or sets of records as the means to do the comparison.
- JDBTools [Shaf] is a Java tool set which can be used to detect differences between two databases, move data between them, or synchronize them. It cannot be used for the EPC database because it assumes that each row in a database table is identified by a unique number.

Database Replication

Database replication techniques from IBM, Oracle, Sybase, and Praxis International were presented in section 4.3.

The groupware application Lotus Notes offers the feature to replicate its document database [Moo95].

Research on Replication Algorithms

There is also some research concerning data replication algorithms (e.g. [BK97], [WJH97], [AZ93], [WP93]), but the focus is on consistency, reliability, and automatic conflict resolution techniques, not on efficient use of the network.

List of Figures

List of Tables

Bibliography

[UML97] *UML Notation Guide*. Rational Software Corporation, September 1997. http://www.rational.com/uml

[Fow97] Martin Fowler, Kendal Scott: UML konzentriert. Addison-Wesley-Longman, 1997.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides: *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[BMR+98] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad und Michael Stal: *Patternorientierte Software-Architektur. Ein Pattern-System*. Addison-Wesley-Longman, 1998.

[Meh] Peter C. Mehlitz: *Using Patterns in Java*. TransVirtual Technologies, Inc. http://www.transvirtual.com/users/peter/patterns

[Tho97] Charles Thompson: *Database Replication - Comparing Three Leading DBMS Vendors' Approaches to Replication*. In: DBMS, May 1997, Volume 10 Number 5. http://www.dbmsmag.com/9705d15.html

[Schu97] Robin Schumacher: *Inside Oracle 8 - Oracle's Latest Release Brings Extended-Relational Technology and Improved Parallel Processing to the Enterprise*. In: DBMS, December 1997, Volume 10 Number 13. http://www.dbmsmag.com/9712d13.html

[Fei97] Andi Feibus: *Databases Hit The Road - Portable versions of enterprise-class databases from Computer Associates, Oracle, and Sybase can help keep remote and mobile users connected to critical data-but not without some work*. In: InformationWeek Labs, November 1997. http://www.informationweek.com/655/55oldbs.htm

[Ren97] Martin Rennhackkamp: *Mobile Database Replication*. In: DBMS, October 1997. http://www.dbmsmag.com/9710d17.html

[Gol95] R. Goldring: *Things every update replication customer should know*. SIGMOD Record (ACM Special Interest Group on Management of Data), 24(2), pp. 439-440, June 1995.

[Froe96] Glenn Froemming: *Design and Replication Issues with Mobile Applications*, Part 1 and 2. In DBMS, March 1996 and April 1996. http://www.dbmsmag.com/9603d13.html, http://www.dbmsmag.com/9604d14.html

[BK97] Yuri Breitbart and Henry F. Korth: *Replication and Consistency: Being Lazy Helps Sometimes*. Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '97), pp. 173-184, Association for Computing Machinery, May 1997.

[WJH97] Ouri Wolfson and Sushil Jajodia and Yixiu Huang: *An Adaptive Data Replication Algorithm*. ACM Transactions on Database Systems, 22(2), pp. 255-314, June 1997.

[AZ93] Swarup Acharya and Stanley B. Zdonik: *An Efficient Scheme for Dynamic Data Replication.* Technical Report, Department of Computer Science, Brown University, Number CS-93-43, September 1993.

[WP93] Q. R. Wang and J.-F. Paris: *Managing Replicated Data Using Referees*. ECOOP'95 Workshop on Mobility and Replication, August 1995.

[McDo] Josh MacDonald: *XDelta*. http://scam.xcf.berkeley.edu/˜jmacd/xdelta.html

[TM96] Andrew Tridgell and Paul Mackerras: *The rsync algorithm*. Technical Report, Department of Computer Science, The Australian National University, Number TR-CS-96-05, June 1996.

[BP98] S. Balasubramaniam and Benjamin C. Pierce: *What is a File Synchronizer?* Indiana University, CSCI Technical report #507, April 1998
http://www.cis.upenn.edu/˜bcpierce/

[Shaf] JDB Tools. Shafir, Inc.
http://www.shafir.com/Products/JDBTools/index.htm

[Kis96] James J. Kistler: *Disconnected Operation in a Distributed File System*. PhD Thesis. Carnegie Mellon University, 1996.
http://www.cs.cmu.edu/afs/cs/project/coda/Web/coda.html

[GHM+90] Richard D. Guy, John Heidemann, Wai Mak, Thomas W. Page, Gerald J. Popek, and Dieter Rothmeiner: *Implementation of the Ficus Replicated File System*. Technical Report, University of California, Los Angeles, 1990.

[RPG+96] P. Reiher, J. Popek, M. Gunter, J. Salomone, and D. Ratner: *Peer-to-peer reconciliation based replication for mobile computers.* In: European Conference on Object Oriented Programming '96, Second Workshop on Mobility and Replication, June 1996.

[CG97] Sudarshan S. Chawathe and Hector Garcia-Molina: *Meaningful Change Detection in Structured Data*. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD Record, Vol. 26,2, pp. 26-37, ACM Press, May 13-15 1997.

[BR97] Daniel Barbara and Sridhar Ramaswamy: *Comparison Techniques for Large Database Replicas*. Abstract: http://www.bell-labs.com/user/sridhar/ftp/dbdiff.abstract

[Moo95] Kenneth Moore: *The Lotus Notes Storage System*. Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, pp. 427-428, 22-25 May 1995.

[Syb98a] *SYBASE Replication Server: A Practical Architecture for Distributing and Sharing Corporate Information*. Sybase, Inc. , 1998.
http://www.sybase.com/products/datamove/repserver_wpaper.html

[Col93] Malcolm Colton: *Replicated data in a distributed environment*. SIGMOD Record (ACM Special Interest Group on Management of Data), 22(2), pp. 464-466, June 1993.

[GWD94] A. Gorelik and Yongdong Wang and M. Deppe: *Sybase Replication Server*. SIGMOD Record (ACM Special Interest Group on Management of Data), 23(2), p. 469, June 1994.

[Ren96] Martin Rennhackkamp: *Sybase SQL Server 11*. In: DBMS, November 1996. http://www.dbmsmag.com/9611d54.html

[Syb98b] *SQL REMOTE: Replication Anywhere*. Sybase, Inc., 1998. http://www.sybase.com/products/anywhere/remote2.html

[Ora98] *Oracle Advanced Replication*. Oracle Corporation, 1998. http://www.oracle.com/st/collateral/html/ar_collateral.html

[DDD+94] D. Daniels and Lip Boon Doo and A. Downing and C. Elsbernd and G. Hallmark and S. Jain and B. Jenkins and P. Lim and G. Smith and B. Souder and J. Stamos: *Oracle's Symmetric Replication Technology and Implications for Application Design*. SIGMOD Record (ACM Special Interest Group on Management of Data), 23(2), pp. 467-467, June 1994

[IBM98a] Data Management White Papers. IBM Corporation, 1998. http://www.software.ibm.com/data/pubs/papers/

[IBM98b] MQSeries: Message Oriented Middleware. IBM Corporation, 1998. http://www.software.ibm.com/ts/mqseries/library/whitepapers/mqover/

[Pra98] *OmniReplicator - Heterogeneous, Bi-Directional Data Replication*. Praxis International Inc. http://www.praxisint.com/omnirepx.html

[Tra98] TransbaseCD Product Information. Transaction Software GmbH. http://www.transaction.de/products/tbcd-fs.htm

[JDBC98] JDBC - Connecting Java and Databases. http://www.javasoft.com/products/jdk/1.1/docs/guide/jdbc/index.html

[JRef98] Java Reflection. http://www.javasoft.com/products/jdk/1.1/docs/guide/reflection/index.html

[JOS98] Java Object Serialization and Externalization. http://www.javasoft.com/products/jdk/1.1/docs/guide/serialization/index.html

[JMS98] Java Message Service http://www.javasoft.com/products/jms/index.html

[JCC98] Java Code Conventions. http://www.javasoft.com/docs/codeconv/html/CodeConventionsTOC.doc.html

[MAD98] Mercedes Aftersales Database MAD. Mercedes internal document.

[PDV98] STAR PARTS Datenversorgung Version 1.0. AS02 Anforderungskatalog und AS05 Systemkurzbeschreibung. Mercedes internal document.

[ELD97] Interface ELDAS to EPC, Data sets description, Programme description. Mercedes internal document.

Appendix A

Electronic Parts Catalogues database

A.1 Tables of StarParts Schema

The following table shows all tables of the StarParts database schema used within the StarNetwork database StarDB along with the class name used within this thesis and a short description.

table name	class/association	cat. section	description
KATALOG	model catalog	model	vehicle classes and area a catalog is used for
BAUMUSTER	model	model	name and description of model
FGST_AGG	model–model	model	usable aggregate models for chassis model
AGG_FGST	model–model	model	usable chassis models for aggregate model
KG	construction group	model	name of construction group
BT_NAME_D	image group	model	name of image group in german
BT_NAME_E	image group	model	same in english
BT_NAME_F	image group	model	same in french
BT_NAME_S	image group	model	same in spanish
BT_NAME_P	image group	model	same in Portuguese
BT_NAME_I	image group	model	same in italian
BILDTAFEL	image identifier	model	reference to image file and release date
MAPS	image number	model	image number contained in an image
TPOS	part record	model	use conditions and amounts for a part
FUSSNOTE	footnote	model	footnote text
FUNO_TAB	table footnote	model	text of grouped footnote
SA_VERWENDUNG	cons. group–stroke vers.	model !	stroke versions usable w/construction group
SA_BEN	special version catalog	sp. version	vehicle classes, area, and name of sp. version
SA_TITEL	stroke version	sp. version	title of stroke version
SA_BILDTAFEL	sp. vers. image identifier	sp. version	reference to image file and release date
SA_MAPS	sp. vers. image number	sp. version	image number contained in an image
SA_TPOS	sp. vers. part record	sp. version	use conditions and amounts for part
SA_FUSSNOTE	sp. vers. footnote	sp. version	footnote text
SA_FUNO_TAB	sp. vers. table footnote	sp. version	text of grouped footnote
SA_UBM	special version–model	sp. version	models which may contain special version
BEITEXT	supplementary text	suppl. text	supplementary text
CODE	code table	-	code table to calculate DNF_CODE in TPOS

A.2 Area-country-codes

These are the codes used for the area country code contained in the field BER_CODE of model catalogues (table KATALOG) and special version catalogues (table SA_BEN).

code	description	code	description
A	MBA	N	Mexico
B	MBB	P	Indonesia
C	FFB/FAP Beograd Yugoslavia	Q	MBB-Industrial Engines
D	MB Spain	R	MBTC/USA
E	ADE South Africa	S	Japan
F	MBNA USA	U	Australia
G	Otomarsan Turkey	W	NAW Switzerland
H	Iran (Civil)	1	General Literature
I	MIO Iran (Iran Army)	2	Catalogue 980/981
J	MB Spares, South Africa	3	AGGR.LIT Truck-Platform
K	MBSA	5	Category-SA
L	ASTAS South Africa	6	Special literature not on MF
M	Nigeria	7	Special literature

A.3 Vehicle Class Codes

These are the the codes contained in the attribute SORT_KLASSE used to specify which vehicle classes a model catalog (table KATALOG) or a special version catalog (table SA_BEN) is valid for. Some catalogues are used for more than one vehicle class, especially aggregate model and special version catalogues. The column position contains the index used for the class tag in the SORT_KLASSE attribute.

position	tag	vehicle class
1	P	Car
2	G	G Wagon
3	L	Light Transporter
4	T	Transporter
5	F	Light Truck
6	M	Medium Truck
7	S	Heavy Truck
8	O	Bus
9	U	Unimog
10	K	MB-Trac
11	E	Industrial Engines

A.4 Attributes in the EPC database

The following table list all attributes contained in the EPC database schema along with a translation and a short description. The column ’key’ tells whether the attribute is used as a primary key (P) or foreign key (F) in some tables.

attribute	short description	used in tables	key	description
ADR_ERG_TEXT	suppl. text id	BEITEXT,(SA_)TPOS	P/F	identifier for supplementary text
AGG_ART	aggregate type	FGST_AGG	-	motor, transmission, axis, ...
AGG_BM	aggregate model	FGST_AGG,AGG_FGST	P	aggregate model identifier part two
AGG_BR	aggregate line	FGST_AGG,AGG_FGST	P	aggregate model identifier part one
ALLE_MENGEN	all quantities	(SA_)TPOS	-	quantities of part in models / str. versions
ANZ_BT	image count	SA_BEN	-	number of images for sp. vers. catalog
AN_KZ	change/new flag	(SA_)TPOS	-	part record was changed or is new
ART	model type	BAUMUSTER	-	aggregate or chassis
ART	type flag	TPOS	-	flag: component fields KP_... filled in
BEN_X	name	KG	-	name of construction group
BER_CODE	area code	KATALOG,SA_BEN	-	area or country catalog is used for
BESCHR_X	description	BAUMUSTER	-	model descriptions
BILDTAFEL_NAME	image name	SA_BILDTAFEL	-	name of special version image
BILD_NR	image number	(SA_)MAPS,(SA_)TPOS	P/F	image number of part shown
BLOCK_ZAEHLER	block counter	(SA_)FUNO_TAB	P	used to group footnotes to block=table
BLOCK_ZEILE	block line no.	(SA_)FUNO_TAB	P	line number within block (redundant)
BM	model line	BAUMUSTER	P	model identifier part two
BR	model	BAUMUSTER,SA_UBM	P	model identifier part one
BR_KATEGORIE	category	KATALOG	-	constant ’BR-Kategorie’ (unused)
BT_ANZ	image count	KG	-	number of images per construction group
BT_IDENT	image identifier	BILDTAFEL,MAPS	P	unique image identifier, part of filename
BT_KENN_ZIFFER	image flag	SA_BEN	-	show images before or between parts ?
BT_NAME	image group title	BT_NAME_X	-	title of image group
BT_NR	image file no.	BILDTAFEL,(SA_)TPOS	-	temporary used until image group known
CODE_BDG	condition code	TPOS,SA_BEN	-	condition codes for usage
DATUM	date	BEITEXT	-	julian date (YYYYDDD)
DATUM	date	(SA_)BILD.,(SA_)MAPS	-	release date of illustration
DNF_CODE	code	TPOS	-	experimental effective code representation
EINRUECKZAHL	indentation	(SA_)TPOS	-	show text indented
ENTF_TNR	remove part no.	SA_TPOS	-	part number from model catalog replaced
ERG_TEXT_X	suppl. text	BEITEXT	-	supplementary text
ERSETZT_KZ	replace flag	(SA_)TPOS	-	part number of part record was replaced
FGST_BEZ	chassis name	AGG_FGST	-	name of chassis
FGST_BM	chassis model	FGST_AGG,AGG_FGST	P	chassis model identifier part two
FGST_BR	chassis line	FGST_AGG,AGG_FGST	P	chassis model identifier part one
FLAG	image flag	KG	-	show images before or between parts ?
FN[1-6]	footnote	(SA_)TPOS,SA_TITEL	F	reference to footnote number FN_NR
FN_ART	footnote type	all footnote	-	’9’, ’X’, or ’Y’ (from conversion)
FN_FOLGE	fn. sequence	all footnoe	P	footnote line identifier part one

attribute	short description	used in tables	key	description
FN_NR	footnote number	all footnote	P	footnote number referenced
FN_SPRACHE	fn. language	all footnote	P	language of footnote line
FN_TEXT	footnote text	all footnote	-	one line of footnote text
FN_ZEILE	footnote line	all footnote	P	footnote line identifier part two
FOLGE	sequence	BAUMUSTER	-	redundant line identifier
H_BEN_X	main name	(SA_)TPOS	-	main name of part
INTERV	interval	SA_TPOS	P	stroke version interval of part record
KAT_NR	catalog id	all model	P	model catalog identifier (3 character)
KG	construct. group	most model	P	construction group identifier (2 digit)
KP_AUSF[1-5]_ALL	stroke versions	TPOS	-	up to 9 stroke versions per component
KP_RUMPF[1-5]	components	TPOS	-	component for part record
LENK_LL	left hand drive	SA_TPOS	-	part is for left hand drive
LENK_LR	right hand drive	SA_TPOS	-	part is for right hand drive
LFD_NR	running number	(SA_)TPOS	P	running number of part record
LG_LA	left/automatic	TPOS	-	valid for left hand drive, automatic trans.
LG_LM	left/manual	TPOS	-	valid for left hand drive, manual trans.
LG_RA	right/automatic	TPOS	-	valid for right hand drive, automatic trans.
LG_RM	right/manual	TPOS	-	valid for right hand drive, manual trans.
NAME	sp. vers. name	SA_BEN	-	name of special version
NAME_X	stroke vers. name	SA_TITEL	-	name of stroke version
POS_NR	position index	BAUMUSTER	P	index of model in TPOS.ALLE_MENGEN
PREFIX	image prefix	BILDTAFEL,MAPS	-	prefix of image filename
REP_TNR_ALL	replace parts	(SA_)TPOS	-	part number(s) that replace part
RUMPF	main number	all sp. vers.	P	six digit special version main number
SORT_KLASSE	vehicle class tags	KATALOG,SA_BEN	-	vehicle classes catalog is valid for
SPRACHE	language	SA_BEN	P	language of NAME column
SPR_NEUTR_TXT	lang. neutr. text	(SA_)TPOS	-	language neutral text in part record
STRICH_AUSF	stroke version	most sp. vers.	P	2 digit stroke version identifier
TNR	part number	(SA_)TPOS	-	part number record describes
TU	image group	some model	P/F	3 digit image group identifer
UBM	sub-models	SA_UBM	-	stroke version usable with these models
VERK_BEZ	model name	BAUMUSTER	-	short model name
V_SA_ALL	connect sp. vers.	SA_TITEL	-	connected special versions
WW_KZ	interchangeable	(SA_)TPOS	-	part is interchangeable with other parts
WW_TNR_ALL	other parts	(SA_)TPOS	-	part is interchangeable with these parts

A.5 SQL Schema Definition

These are the supplied SQL table definitions of the StarParts (EPC) database for DB2. The german comments have been removed.

```
create table parts.katalog (
  kat_nr                CHAR(3) NOT NULL,
  sort_klasse           CHAR(15),
  ber_code              CHAR(1),
  br_kategorie          VARCHAR(25),
PRIMARY KEY (
  kat_nr)
)
  IN PARTS;

create table parts.baumuster (
   kat_nr               CHAR(3) NOT NULL,
   br                   CHAR(3) NOT NULL,
   bm                   CHAR(3) NOT NULL,
   pos_nr               SMALLINT NOT NULL,
   art                  CHAR(1),
   zeile                CHAR(2) NOT NULL,
   folge                CHAR(1),
   verk_bez             VARCHAR(50),
   beschr_d             VARCHAR(80),
   beschr_e             VARCHAR(80),
   beschr_f             VARCHAR(80),
   beschr_s             VARCHAR(80),
   beschr_p             VARCHAR(80),
   beschr_i             VARCHAR(80),
PRIMARY KEY (
  br, bm, kat_nr, pos_nr, zeile)
)
  IN PARTS;

create table parts.fgst_agg (
   kat_nr               CHAR(3) NOT NULL,
   fgst_br              CHAR(3) NOT NULL,
   fgst_bm              CHAR(3) NOT NULL,
   agg_art              CHAR(2),
   agg_br               CHAR(3) NOT NULL,
   agg_bm               CHAR(3) NOT NULL,
PRIMARY KEY (
  kat_nr, fgst_br, fgst_bm, agg_br, agg_bm)
)
  IN PARTS;

create table parts.agg_fgst (
   kat_nr               CHAR(3) NOT NULL,
   agg_br               CHAR(3) NOT NULL,
   agg_bm               CHAR(3) NOT NULL,
   fgst_br              CHAR(3) NOT NULL,
   fgst_bm              CHAR(3) NOT NULL,
   fgst_bez             VARCHAR(512),
PRIMARY KEY (
  kat_nr, agg_br, agg_bm, fgst_br, fgst_bm)
)
  IN PARTS;

create table parts.kg (
  kat_nr                CHAR(3) NOT NULL,
  kg                    CHAR(2) NOT NULL,
  bt_anz                INTEGER,
  flag                  CHAR,
  ben_d                 VARCHAR(40),
  ben_e                 VARCHAR(40),
  ben_f                 VARCHAR(40),
  ben_s                 VARCHAR(40),
  ben_p                 VARCHAR(40),
  ben_i                 VARCHAR(40),
PRIMARY KEY (
  kat_nr, kg)
)
  IN PARTS;

create table parts.bt_name_d (
  kat_nr                CHAR(3) NOT NULL,
  kg                    CHAR(2) NOT NULL,
  tu                    CHAR(3) NOT NULL,
  bt_name               VARCHAR(95) NOT NULL,
PRIMARY KEY (
  kat_nr, kg, tu)
)
  IN PARTS;

create table parts.bt_name_e (
  kat_nr                CHAR(3) NOT NULL,
  kg                    CHAR(2) NOT NULL,
  tu                    CHAR(3) NOT NULL,
  bt_name               VARCHAR(95) NOT NULL,
PRIMARY KEY (
  kat_nr, kg, tu)
)
  IN PARTS;

create table parts.bt_name_f (
  kat_nr                CHAR(3) NOT NULL,
  kg                    CHAR(2) NOT NULL,
  tu                    CHAR(3) NOT NULL,
  bt_name               VARCHAR(95) NOT NULL,
PRIMARY KEY (
  kat_nr, kg, tu)
)
  IN PARTS;

create table parts.bt_name_s (
  kat_nr                CHAR(3) NOT NULL,
  kg                    CHAR(2) NOT NULL,
  tu                    CHAR(3) NOT NULL,
  bt_name               VARCHAR(95) NOT NULL,
PRIMARY KEY (
  kat_nr, kg, tu)
)
  IN PARTS;

create table parts.bt_name_p (
  kat_nr                CHAR(3) NOT NULL,
  kg                    CHAR(2) NOT NULL,
  tu                    CHAR(3) NOT NULL,
  bt_name               VARCHAR(95) NOT NULL,
PRIMARY KEY (
  kat_nr, kg, tu)
)
  IN PARTS;

create table parts.bt_name_i (
  kat_nr                CHAR(3) NOT NULL,
  kg                    CHAR(2) NOT NULL,
  tu                    CHAR(3) NOT NULL,
  bt_name               VARCHAR(95) NOT NULL,
PRIMARY KEY (
  kat_nr, kg, tu)
)
  IN PARTS;

create table parts.bildtafel (
  kat_nr                CHAR(3) NOT NULL,
  kg                    CHAR(2) NOT NULL,
  tu                    CHAR(3) NOT NULL,
  bt_ident              CHAR(7) NOT NULL,
  bt_nr                 CHAR(2) NOT NULL,
  prefix                CHAR(2) NOT NULL,
  datum                 DATE,
PRIMARY KEY (
  kat_nr, kg, tu, bt_ident)
)
  IN PARTS;

create table parts.maps (
  kat_nr                CHAR(3) NOT NULL,
  kg                    CHAR(2) NOT NULL,
  tu                    CHAR(3) NOT NULL,
  bt_ident              CHAR(7) NOT NULL,
  bild_nr               CHAR(5) NOT NULL,
  prefix                CHAR(2),
  datum                 DATE,
PRIMARY KEY (
  kat_nr, kg, tu, bt_ident, bild_nr)
)
  IN PARTS;

create table parts.tpos (
  kat_nr                CHAR(3) NOT NULL,
  kg                    CHAR(2) NOT NULL,
  lfd_nr                INTEGER NOT NULL,
  bt_nr                 CHAR(2) NOT NULL,
  bild_nr               CHAR(5),
  ersetzt_kz            CHAR(1),
  h_ben_d               VARCHAR(15),
  h_ben_e               VARCHAR(25),
  h_ben_f               VARCHAR(25),
  h_ben_s               VARCHAR(25),
  h_ben_p               VARCHAR(25),
  h_ben_i               VARCHAR(25),
  adr_erg_text          CHAR(11),
  spr_neutr_txt         VARCHAR(40),
  einrueckzahl          CHAR(1),
  ww_kz                 CHAR(1),
  an_kz                 CHAR(1),
  lg_lm                 CHAR(1),
  lg_la                 CHAR(1),
  lg_rm                 CHAR(1),
  lg_ra                 CHAR(1),
  fn1                   SMALLINT,
  fn2                   SMALLINT,
  fn3                   SMALLINT,
  fn4                   SMALLINT,
  fn5                   SMALLINT,
  fn6                   SMALLINT,
  tnr                   CHAR(19),
  alle_mengen           VARCHAR(66),
  art                   CHAR(3),
  kp_rumpf1             CHAR(6),
  kp_ausf1_all          CHAR(18),
  kp_rumpf2             CHAR(6),
  kp_ausf2_all          CHAR(18),
  kp_rumpf3             CHAR(6),
  kp_ausf3_all          CHAR(18),
  kp_rumpf4             CHAR(6),
  kp_ausf4_all          CHAR(18),
  kp_rumpf5             CHAR(6),
  kp_ausf5_all          CHAR(18),
  code_bdg              VARCHAR(60),
  rep_tnr_all           VARCHAR(604),
  ww_tnr_all            VARCHAR(456),
  dnf_code              VARCHAR(1000),
  tu                    CHAR(3),
PRIMARY KEY (
  kat_nr, kg, lfd_nr)
)
  IN PARTS;


create table parts.fussnote (
  kat_nr                CHAR(3) NOT NULL,
  kg                    CHAR(2) NOT NULL,
  fn_nr                 SMALLINT NOT NULL,
  fn_folge              SMALLINT NOT NULL,
  fn_zeile              SMALLINT NOT NULL,
  fn_sprache            CHAR(1) NOT NULL,
  fn_art                CHAR(1),
  fn_text               VARCHAR(130),
PRIMARY KEY (
   kat_nr, kg, fn_nr, fn_folge, fn_zeile, fn_sprache)
)
  IN PARTS;

create table parts.funo_tab (
  kat_nr                CHAR(3) NOT NULL,
  kg                    CHAR(2) NOT NULL,
  block_zaehler         SMALLINT NOT NULL,
  block_zeile           SMALLINT NOT NULL,
  fn_nr                 SMALLINT NOT NULL,
  fn_folge              SMALLINT NOT NULL,
  fn_zeile              SMALLINT NOT NULL,
  fn_sprache            CHAR(1),
  fn_art                CHAR(1),
  fn_text               VARCHAR(130),
PRIMARY KEY (
   kat_nr, kg, block_zaehler, block_zeile)
)
  IN PARTS;

create table parts.sa_verwendung (
  kat_nr                CHAR(3) NOT NULL,
  kg                    CHAR(2) NOT NULL,
  rumpf                 CHAR(6) NOT NULL,
  strich_ausf           SMALLINT NOT NULL,
PRIMARY KEY (
  kat_nr, kg, rumpf, strich_ausf )
)
  IN PARTS;

create table parts.sa_ben (
  rumpf                 CHAR(6) NOT NULL,
  zeile                 SMALLINT NOT NULL,
  sprache               CHAR(1) NOT NULL,
  anz_bt                CHAR(2),
  bt_kenn_ziffer        CHAR(1),
  ber_code              CHAR(1),
  sort_klasse           CHAR(15),
  name                  VARCHAR(140),
  code_bdg              VARCHAR(200),
PRIMARY KEY (
  rumpf, zeile,sprache)
)
  IN PARTS;

create table parts.sa_titel (
  rumpf                 CHAR(6) NOT NULL,
  strich_ausf           SMALLINT NOT NULL,
  zeile                 SMALLINT NOT NULL,
  name_d                VARCHAR(50),
  name_e                VARCHAR(50),
  name_f                VARCHAR(50),
  name_s                VARCHAR(50),
  name_p                VARCHAR(50),
  name_i                VARCHAR(50),
  fn1                   SMALLINT,
  fn2                   SMALLINT,
  fn3                   SMALLINT,
  fn4                   SMALLINT,
  fn5                   SMALLINT,
  v_sa_all              VARCHAR(120),
PRIMARY KEY (
  rumpf, strich_ausf, zeile)
)
  IN PARTS;

create table parts.sa_bildtafel(
  rumpf                 CHAR(6) NOT NULL,
  strich_ausf           SMALLINT NOT NULL,
  bildtafel_name        CHAR(14) NOT NULL,
  datum                 DATE,
PRIMARY KEY (
  rumpf, strich_ausf, bildtafel_name)
)
  IN PARTS;

create table parts.sa_maps (
  rumpf                 CHAR(6) NOT NULL,
  strich_ausf           SMALLINT NOT NULL,
  bild_nr               CHAR(5) NOT NULL,
  datum                 DATE,
PRIMARY KEY (
  rumpf, strich_ausf, bild_nr)
)
  IN PARTS;

create table parts.sa_tpos (
  rumpf                 CHAR(6) NOT NULL,
  interv                SMALLINT NOT NULL,
  lfd_nr                INTEGER NOT NULL,
  bild_nr               CHAR(5),
  bt_nr                 CHAR(2),
  ersetzt_kz            CHAR(1),
  tnr                   CHAR(19),
  entf_tnr              CHAR(19),
  h_ben_d               CHAR(15),
  h_ben_e               CHAR(25),
  h_ben_f               CHAR(25),
  h_ben_s               CHAR(25),
  h_ben_p               CHAR(25),
  h_ben_i               CHAR(25),
  adr_erg_text          CHAR(11),
  spr_neutr_txt         VARCHAR(40),
  einrueckzahl          CHAR(1),
  ww_kz                 CHAR(1),
  an_kz                 CHAR(1),
  lenk_ll               CHAR(1),
  lenk_lr               CHAR(1),
  fn1                   SMALLINT,
  fn2                   SMALLINT,
  fn3                   SMALLINT,
  fn4                   SMALLINT,
  fn5                   SMALLINT,
  fn6                   SMALLINT,
  alle_mengen           VARCHAR(30),
  rep_tnr_all           VARCHAR(604),
  ww_tnr_all            VARCHAR(456),
PRIMARY KEY (
  rumpf, interv, lfd_nr)
)
  IN PARTS;

create table parts.sa_fussnote (
  rumpf                 CHAR(6) NOT NULL,
  fn_nr                 SMALLINT NOT NULL,
  fn_folge              SMALLINT NOT NULL,
  fn_zeile              SMALLINT NOT NULL,
  fn_sprache            CHAR(1) NOT NULL,
  fn_art                CHAR(1),
  fn_text               VARCHAR(130),
PRIMARY KEY (
   rumpf, fn_nr, fn_folge, fn_zeile, fn_sprache)
```

```
)
  IN PARTS;

create table parts.sa_funo_tab (
  rumpf                 CHAR(6) NOT NULL,
  block_zaehler         SMALLINT NOT NULL,
  block_zeile           SMALLINT NOT NULL,
  fn_nr                 SMALLINT NOT NULL,
  fn_folge              SMALLINT NOT NULL,
  fn_zeile              SMALLINT NOT NULL,
  fn_sprache            CHAR(1),
  fn_art                CHAR(1),
  fn_text               VARCHAR(130),
PRIMARY KEY (
   rumpf, block_zaehler, block_zeile)
)
  IN PARTS;

create table parts.sa_ubm (
  rumpf                 CHAR(6) NOT NULL,
  strich_ausf           SMALLINT NOT NULL,
  br                    CHAR(3) NOT NULL,
  ubm                   VARCHAR(480),
PRIMARY KEY (
  rumpf, strich_ausf, br)
)
  IN PARTS;

create table parts.beitext (
  adr_erg_text          CHAR(11) NOT NULL,
  datum                 CHAR(7),
  erg_text_d            VARCHAR(120),
  erg_text_e            VARCHAR(120),
  erg_text_f            VARCHAR(120),
  erg_text_s            VARCHAR(120),
  erg_text_p            VARCHAR(120),
  erg_text_i            VARCHAR(120),
PRIMARY KEY (
  adr_erg_text)
)
  IN PARTS;


CREATE TABLE parts.code (
  code                VARCHAR(7) NOT NULL,
PRIMARY KEY (code)
)
  IN PARTS;
```

Appendix B

FDOK database

B.1 SQL Schema Definition

These are the supplied SQL definitions of the StarIdent (FDOK) database for DB2. The german comments have been removed.

```
connect to ident;


 drop table ident.fdk;
 create table ident.fdk (
    whc             CHAR(3) NOT NULL,
    fin_rumpf       CHAR(14) NOT NULL,
    vin             VARCHAR(20),
    motor           VARCHAR(18),
    getriebe        VARCHAR(18),
    auftrag         CHAR(10),
    versorg_dat     timestamp DEFAULT CURRENT TIMESTAMP,
    object          VARCHAR(3888) FOR BIT DATA,
    -- Timestamp, an dem dieses Tupel von FDOK versorgt wurde
 PRIMARY KEY (
    fin_rumpf, whc))  in
  IDENT;


 drop table ident.fdk_sa_ben;
 create table ident.fdk_sa_ben (
    rumpf               CHAR(6) NOT NULL,
    ben_d               VARCHAR(140),
    ben_e               VARCHAR(140),
    ben_f               VARCHAR(140),
    ben_s               VARCHAR(140),
    ben_p               VARCHAR(140),
    ben_i               VARCHAR(140),
    reserve1            VARCHAR(140),
    reserve2            VARCHAR(140),
    sort_klasse         CHAR(15),
    versorg_dat         timestamp DEFAULT CURRENT TIMESTAMP,
    -- Timestamp, an dem dieses Tupel von FDOK versorgt wurde
 PRIMARY KEY (
    rumpf)) in
  IDENT;
 drop table ident.fdk_code_ben_nfz;
 create table ident.fdk_code_ben_nfz (
    code_nr             CHAR(3) NOT NULL,
    sparte              CHAR(1) NOT NULL,
    ben_d               VARCHAR(70) ,
    ben_e               VARCHAR(70) ,
    ben_f               VARCHAR(70) ,
    ben_i               VARCHAR(70) ,
    ben_s               VARCHAR(70) ,
    ben_p               VARCHAR(70) ,
    versorg_dat         timestamp DEFAULT CURRENT TIMESTAMP,
    -- Timestamp, an dem dieses Tupel von FDOK versorgt wurde
 PRIMARY KEY (
    code_nr,sparte)) in
  IDENT;


drop table ident.fdk_code_ben_pkw;
create table ident.fdk_code_ben_pkw (
   code_nr              CHAR(4) NOT NULL,
   datum_von            timestamp DEFAULT CURRENT TIMESTAMP NOT NULL,
   datum_bis            timestamp DEFAULT CURRENT TIMESTAMP NOT NULL,
   ben_d                VARCHAR(70),
   ben_e                VARCHAR(70),
   ben_f                VARCHAR(70),
   ben_s                VARCHAR(70),
   ben_i                VARCHAR(70),
   versorg_dat          timestamp DEFAULT CURRENT TIMESTAMP,
 PRIMARY KEY (
    code_nr, datum_von, datum_bis)) in
  IDENT;
```