Open Source Database Systems: Systems study, Performance and Scalability

Toni Strandell

Helsinki 8th May 2003

Master's Thesis

UNIVERSITY OF HELSINKI
Department of Computer Science

Faculty of Science | Department of Computer Science

Tekijä — Författare — Author
Toni Strandell

Työn nimi — Arbetets titel — Title

Open Source Database Systems: Systems study, Performance and Scalability

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's Thesis	8th May 2003	54 pages + appendix 3 pages

Tiivistelmä — Referat — Abstract

Database systems have been studied over three decades. The methods used in modern systems are well understood. The movement in the Open Source community and the development of open source database systems give subject to new considerations. The source codes of the products are public and their non-commercial use is free, whereas the prices of commercial databases are very high. A free or inexpensive open source database could be a real option to many enterprises if only the functionality of it would meet today's high requirements. High performance, support for transactions, automatic recovery and scalability are the basic requirements. High availability is often desired as well.

This thesis presents the theory behind transaction management, storage structures, concurrency control and availability. The open source database market is surveyed and some open source database systems are selected for further investigating and experimenting. The techniques and methods used in the databases are presented and compared to the theory. The experimenting is done in a mobile telecommunications environment, which is a modern business area involving a large amount of data processing. The experiments are done on a Linux platform.

ACM Computing Classification System (CCS):
H.1.1 [Systems and Information Theory]
H.2 [Database Management]
H.3.4 [Systems and Software]

Avainsanat — Nyckelord — Keywords
database systems, database management, open source databases, performance evaluation

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Contents

1 Introduction

Database systems have been thoroughly studied over the past 30 years and various commercial database systems implement the well investigated theories. These systems provide the user with sophisticated concurrency control and transaction management, backup, replication, and recovery schemes. However, the prices of the commercial databases are thousands or even tens of thousands of euros. This can be unacceptably high for small and medium size enterprises.

As an alternative to the commercial database systems open source databases exist. The idea behind the open source is that the product's source and binary codes can be obtained for free. The product's license usually says that the usage is free in non-commercial systems and in products that are published under the same license. It is a common practice to sell licenses also for commercial use. The prices of the licenses usually are a lot lower than the prices of commercial systems. Inexpensive open source databases could be a tempting option for many enterprises if only they offered functionality and performance comparable to those of commercial systems.

The business model of the companies behind the open source databases is also quite different from the traditional one. Traditionally software products are protected by all available means, the source code above all. The revenue comes mainly from selling licenses to use the products, whereas with open source all the source code is available publicly. Usually the companies offer customer service and product support for a fee and sell licenses for commercial use of the products. These are the main sources of the revenue. The development of the products is global and the Internet is used as the market place and distribution channel.

Mobile phone markets are worldwide and more people use mobile phones in everyday communication. With Third Generation mobile phones and mobile phone networks the area is evolving. The mobile telecommunications environment involves a large amount of data processing and transferring. Traditionally data management and data storage employ embedded and commercial database systems but with the Third Generation mobile phone networks this could change. Open source database systems might be a realistic alternative to commercial systems if only they supported the functionality needed by the mobile telecommunications systems.

In this thesis we will survey the open source database markets and introduce some of the available open source databases. Some of the presented databases are selected for further investigation and experimentation. The design and implementation conventions of the selected databases are presented and compared to the well studied implementation techniques of database systems covered in the literature.

The selection is made on requirements set by the mobile telecommunications environment. The system has to be open source. It has to have commercial support, i.e. at least one company that offers support services and possibly is the developer of the system as well. Tools to build a highly available system are required. In practice availability is increased by replicating the database. Transactions and serializability have to be guaranteed and recovery procedures are required. The system

must not depend on any single operating system or hardware configuration. Furthermore, there must not be any heavy reorganizations of the storage structure or other administrative tasks that disturb normal database operation.

Benchmarks can be used to evaluate the performance of database systems. Generic benchmarks exist, but they are not wellsuited for evaluating the performance in a telecom environment. Home location register (HLR) [ETS] is a database that mobile network operators use to store information about the network's users. In this thesis a benchmark that mimics the home location register and its utilization [Tik02] is used to evaluate the performance and applicability of the selected open source databases. The benchmark contains the most essential parts of a functional HLR system. However, it excludes administrative transactions that constitute only a fraction of the overall load of such a system. Several different read and write transactions typical to a HLR are executed on the databases using a Linux-based testing environment. A commercial database is also included to compare its performance to the selected open source databases. The commercial database is studied in less detail than the open source databases, because it is only used as a reference.

The experiments show that the there exist open source databases, performance characteristics of which are adequate for mobile telecommunications use. However, none of them has all the required features to build a reliable and available system. The open source database market is growing and developing. Open source databases offer alternatives to traditional commercial databases. However, they are not yet mature enough for critical applications.

The rest of this thesis is organized as follows. In Chapter 2 we examine the architecture and implementation techniques of database management systems. The access methods, buffer management policies and join algorithms are presented, as well as concurrency control, transaction management, recovery and availability schemes. In Chapter 3 we introduce some open source databases and evaluate their applicability. In Chapters 4 and 5 we describe the implementation techniques used in the selected open source databases. The implementations are compared to those presented in Chapter 2. In Chapter 6 we report the results of our experiments on performance and scalability of the database systems. The experiments are done with InnoDB and Berkeley DB table handlers under MySQL.

2 Concepts on Database Management

The basic concepts of current database systems are well studied. The most widely used access methods are B^+-trees and hashing [BU77, Com79, Sal96]. In B^+-trees the index information is stored in the inner nodes and the actual data or pointers to the data in the leaf nodes of the tree. Hashing offers fast translation of keys into physical addresses. Concurrency control is normally managed with *two phase locking* (2PL) and *latch coupling* [Moh90, ML92, Tho98]. Alternatives include timestamp-based methods [SKS97], optimistic concurrency control [SKS97, Tho98] and multiversioning [BG83, Tho98].

Transaction management has to fulfill the *ACID properties*, i.e. the transactions have to be atomic, consistent, isolated and their effects durable. This is guaranteed with concurrency control, logging and recovery schemes. Availability is increased with replication [LLSG92] or global locking. Copies of the database are to guarantee that in case of failure the service does not stop. The objective is to achieve high availability: system available 99,999% of the time, i.e. downtime 5 min/year [GR93, p. 97].

2.1 Components of a Database Management System

The architecture of database management systems (DBMSs) vary from product to product. Thus, it is not possible to present a structure that would characterize them all. However, most of the DBMSs follow some basic guidelines and have common components. The *database manager* (Figure 1) consists of the part of a DBMS that accepts queries and places requests to a *file manager* responsible for retrieving the requested data from the secondary storage, usually the hard disk. The data is placed in main memory at least for the duration of its processing. *Buffer manager* transfers the data between main memory and secondary storage. *Transaction manager*, *scheduler* and *recovery manager* together with the buffer manager allow concurrent execution of transactions while preserving the ACID-properties. The operation of the components drawn with solid lines in Figure 1 are explained in the following subsections.

2.2 Data Management

Database systems are to retrieve a given datum from all the data in the database as fast as possible and consuming as little system resources (disk I/Os and memory transfers) as possible. To do this, the data has to be well organized in specific data structures that offer the desired data-handling functionality. Data storage is basically a pool of data pages. The actual data, the tuples (records), are stored on these pages. The objective of access methods (Figure 1) is to offer fast access to these pages. Access methods are the tools to find the desired information in the data file. An access method is constructed from two parts; the access path and the actual data storage.

Generally, the access path is an ordered index or a hash index. Ordered indices are associated with a search key. The keys correspond to the value(s) of some attribute(s) of a relation. In ordered indices the records are accessed performing key comparisons. Hash indices distribute the tuples into buckets based on the hashed values of some attribute(s). Due to this distribution, hash indices are usually non-ordered.

The simplest way to store the data would be to add new tuples to the end of the data file. In this scheme finding a specific tuple would require scanning the data file in sequential order starting from the beginning. On average this would

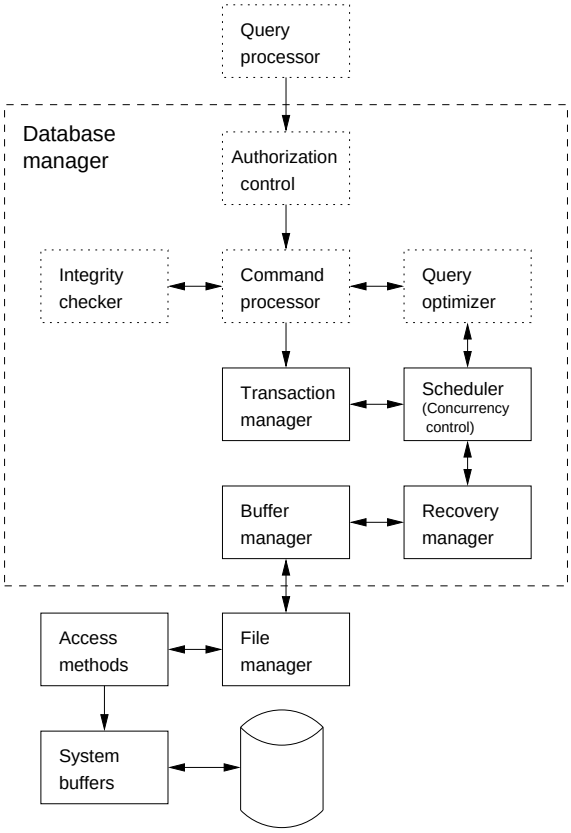

Figure 1: Components of database manager [CB02].

require searching half of the data file every time a single tuple is retrieved from the database. The cost of this scheme is unacceptable in most cases, even considering its simplicity. Accessing the data means retrieving one tuple, a group of tuples based on some criterion, or a whole relation. The purpose of an access method is to translate search keys (attribute values) to tuple identifiers (TIDs) and to find the data pages containing the tuples pointed by the TIDs. Basic query types include retrieving tuples matching the given values and range queries (attribute value between a and b). Different access methods have been developed to avoid searching the whole (or half of the) file in order to find a result to a query.

The most popular access methods in database systems are hashing and B-trees [Sal96]. In hashing the tuples are stored in data pages called buckets identified by the hash value of the search key. Hashing can be static or dynamic, in static hashing bucket overflow has to be handled in some fashion. One way is to chain overflow buckets after the original bucket. However, this deviates from the ideal *one-access retrieval* scheme (only one page needs to be read from the disk). The overflown hash structure can be rehashed into bigger address space. This is a heavy operation that temporarily reduces the level of concurrency dramatically.

In dynamic hashing [Lit80, Lar88], when a collision occurs, more space (buckets) is added to the hash structure and the contents of the overflown bucket are re-hashed

with a new hash function. In some dynamic hashing schemes it is not always the overflown bucket that gets rehashed but instead the buckets are rehashed in some predefined order. Linear hashing [Lit80] is such a method. When collisions occur new buckets are added and the original buckets are rehashed in linear order. The hashing function changes whenever the size of the structure is doubled.

Hashing is very fast in retrieving the tuples that correspond to given attribute values, but performs poorly in range selection queries. The attribute values are distributed across the data file and sequential scan of the entire data file is required.

B-trees [BU77, Com79] are the most utilized structures in databases. There exists a wide variety of B-trees and their naming differs from source to source. The most common B-tree variation in database environment is the one where the access path (index structure) is stored in the inner nodes and the actual data in the leaf nodes. The data in the leaf nodes is ordered by the key and the leaf nodes are chained to allow the data to be accessed sequentially. In many sources and in this thesis this type of structure is called a B^+-tree. Finding the right tuples is based on key comparisons. The searched key value is compared to those found in the index structure thus leading to the right data page. A B^+-tree can be constructed on any group of attributes. The order of the attributes is significant and allows the search to be done with any subset of the attributes starting from the left. The length of the path from the root of the tree to a data page is logarithmic with respect to the number of keys in the tree.

The use of access methods does not overcome the slowness of hard disks, which are usually used as the end storage medium in database systems. Database systems use main memory to buffer data pages allowing faster processing of the data. Ideally all the data in the database fits in the main memory buffer, but usually this is not the case. One of the main goals of database systems is to minimize the number of block transfers between disk and main memory. For simplicity, it is assumed that physical disk blocks and logical database pages are the same size and when a page is written, it is written to the same block on the disk it was read from. Therefore, the names are used here interchangeably.

Buffer manager (Figure 1) is responsible for keeping recently accessed pages in the main memory buffer in case they are accessed again shortly. When a page is requested from the buffer manager, it checks if the page is already in the buffer. If this is the case, the buffer manager returns to the requester the main memory block address containing the page. If the page is not in the buffer, the buffer manager allocates space for the page in the main memory. This might mean throwing out a page and writing the page to disk, if it was modified during its presence in the buffer. The requested block is searched from the disk and read into the buffer block just freed. The block address is returned to the requester.

Deciding which page to remove from the buffer is not a trivial problem. Rather simple and widely implemented strategies in deciding the page to be removed are *first-in-first-out* (FIFO) and *least-recently-used* (LRU) [CB02]. These strategies are very straight-forward and do not take into account the intended use of the pages.

The database system knows the storage structures and algorithms used to calculate, for example, a join of two relations. It can then say or predict which pages of the first relation are needed in which order to join them to the ones of the second relation. The buffer manager can also assume that the data dictionary and index pages are used more frequently than many of the data pages and should not therefore be removed from the buffer unless some other factors strongly indicate otherwise. Despite the fact that the pure LRU strategy does not try to predict how pages are used many database systems use it [SKS97].

When a page is in the buffer, the right tuple has to be located in the page. Current page sizes in databases are large, so that searching a tuple by scanning the whole page is an expensive operation even when the page resides in main memory. Therefore the page contains a page dictionary that helps to find the beginning of a tuple ([GR93] p. 754). The page dictionary contains fixed size offsets of the records from the start of the page. The tuples are stored continuously from the beginning of the page and the page dictionary is stored at the end of the page. A tuple is located scanning only the page dictionary thus saving a lot of calculation.

2.3 Join Algorithms

In databases the join operation combines tuples from two or more relations based on some common information. For example, a banking application could have two relations, customers and accounts. The customers could have many accounts. Then a join could be used to search the accounts of each customer. This result could then be joined with a third relation, events. We could then tell what events (deposits and withdrawals) the customers have completed on each of their accounts. More specifically, a join is used to combine two or more relations when they satisfy a specified *join condition* [ME92]. The join operations have been discussed and studied extensively since they are time-consuming, data-intensive and heavily used in database systems. A two-way join is performed on two relations and a multiway join on more than two [ME92]. A multiway join on n relations produces the same result as $n-1$ two way joins. This is also how a multiway join is usually implemented.

The join operation can be said to be equivalent to a select operation done on a cartesian product. The result of a join is always a subset of the cartesian product performed on the same relations. The result of joining R with n attributes and S with m attributes is a relation Q with $(n+m)$ attributes. A join of relations R and S is written as

$$R \bowtie_{r(a)\theta s(b)} S \equiv \sigma_{r(a)\theta s(b)}[R \times S], r \in R, s \in S.$$

where θ is the join condition that must hold true between the attributes $r(a)$ and $s(b)$ of R and S, respectively. $R \times S$ is a cartesian product of the relations R and S, and σ is a select operation. The θ operator defines the condition that must hold true between the attributes $r(a)$ and $s(b)$ of the relations R and S, respectively. This is the general join called *theta-join*. The result relation Q may then be defined as

$$Q = \{t|t = rs \wedge r \in R \wedge s \in S \wedge t(a)\theta t(b)\}.$$

Q is a subset of a cartesian product of R and S. In the most simple implementation each tuple from R has to be compared to each tuple in S. So the complexity of the join with n tuples in each relation is $O(n^2)$. This requires an immense amount of I/O operations and is far from optimal in most cases. Different algorithms have been developed to reduce the number of comparisons needed, the number of I/O operations needed, or both. Along with the theta-join, several other join types have been defined. These include *equijoin*, *natural join*, *semijoin*, *outerjoin*, *self-join*, *composition* and *division*. Detailed explanations of the join types can be found in [ME92].

Many implementation techniques and methods exist to do a join. The ones discussed below are *nested-loop join*, *sort-merge join* and basic *hash join methods* [ME92, SKS97, CB02]. The simplest of the join methods discussed here is the nested-loop join. One of the joined relations is the inner and the other one the outer relation. Each tuple of the outer relation is compared to every tuple of the inner relation. When the join condition is satisfied, the two tuples are concatenated and written to the output buffer. The algorithm to do the nested-loop join is presented in Figure 2.

```
for each tuple s do
  for each tuple r do
    if r(a)=s(b) then
      concatenate r and s into q
      place q in relation Q
    end if
  end
end
```

Figure 2: The nested-loop join.

To do a cartesian product of two relations, all the data in one of the relations is scanned as many times as the other relation has tuples. This requires a very large amount of disk operations to move the data pages between disk and the main memory buffer. If the buffer can hold another of the relations in it entirely, the number of required disk operations is reduced, but still remains high. Even if the buffer is too small to hold either of the relations, the number of page accesses can still be reduced considerably. The *block nested-loop join* [SKS97] compares the tuples per block basis instead of per tuple as was done in the general nested-loop join. If there exists an index for either of the input relations, it can further reduce the number of block accesses required. In *indexed nested-loop join* [SKS97] the outer relation is scanned and the tuples that satisfy the join condition are found using the index. If indices are available for both relations, selecting the one with lesser tuples is generally more efficient.

The sort-merge [ME92] join is computed in two phases. In the first one the input relations are sorted on the common join attributes. In the second phase both relations are scanned in the order of the join attributes. When the join condition is satisfied, they are concatenated and written to the output buffer. If one or both of the relations are sorted on the join attributes, phase one will be lighter or absent. Figure 3 sketches the algorithm for the sort-merge join. When no indexes on the join attributes exist and there is not enough information to choose a particular join algorithm, the sort-merge join is found to be the best choice in most cases [ME92]. Indexes on the join attributes can be utilized in merge join too. This variant of the method is called *hybrid merge join* [SKS97]. For example, it can take advantage of secondary B^+-tree indexes on the join attributes. In this case the merge is done with the leaf nodes of the index.

```
sort R on r(a)
sort S on s(b)

read first tuple from R
read first tuple from S
for each tuple r do
  while s(b) < r(a)
    read next tuple from S
    if r(a)=s(b) then
      concatenate r and s into q
      place q in Q
    end if
  end
end
```

Figure 3: The sort-merge join.

The hash join methods [Bra84] implement the same basic idea as sort-merge join; a tuple from the first relation is not compared to those from the second relation it cannot possibly join. The implementation, though, is different. The idea of *simple hash join* [ME92] is that tuples r from R and s from S have the same hash values if they satisfy the join condition. But since different keys can have the same hash value, keys of the tuples with the same hash values need to be compared. The simple hash join algorithm is shown in Figure 4.

It might be that the number of available buffer blocks is smaller than the number of desired hash buckets. *Recursive partitioning* [ME92, SKS97] solves this problem. The values are first hashed into equally many partitions as there are buffer blocks and then each of these partitions is hashed into more partitions until the desired amount of partitions is reached. *Hybrid hash join* [Sha86] further optimizes the use of the main memory and reduces disk accesses keeping one of the partitions intact in memory throughout the join calculation.

The nested-loop join is simple and easy to implement and, as a result, widely used

```
for each tuple s in S do
  hash on join attributes s(b)
  place tuples in hash index based on hash values
end

for each tuple r in R do
  hash on join attributes r(a)
  if r hashes to a nonempty bucket of hash index for S then
    if r matches any s in bucket then
      concatenate r and s into q
      place q in Q
    end if
  end if
end
```

Figure 4: The hash join.

even though its performance in many situations is not that good. Hash join methods outperform sort-merge join in many cases [Bra84, Gra94]. The hybrid hash join method [Sha86] is better than sort-merge or the other hash join methods if the relations are sufficiently large. It was improved in [Gra94]. The hash join methods cannot be used for range selection queries, because they do not maintain the order of the tuples.

2.4 Transaction Management

Transaction is a logical unit of work consisting of various operations. The need of transaction management and recovery was noted early and has been studied extensively [Ver78, BHR80, HR83, Reu84]. As a result, a four letter acronym was presented; ACID. A database system must guarantee proper execution of transactions, i.e. the system must maintain the ACID-properties of transactions. *Atomicity* means that all the transaction's operations are executed or none of them is. Each transaction has to transform the database from a consistent state to consistent state when executed as the only transaction in the database (*consistency*). To each transaction the database appears as if it were the only transaction executing in it, thus the transaction is executed in *isolation*. The modifications made by a completed transaction have to be *durable* (persistent) even when system or media failures occur. Ensuring atomicity is the responsibility of the transaction management, durability of the recovery management and isolation of the concurrency control. The application programmer is responsible for the consistency of a single transaction.

A transaction's state changes as it executes. After *beginning* its work it is *active* until it either *commits* or *aborts*. When a transaction is *committed* it has successfully completed its execution and its changes are durable. When a transaction is *aborted*, all the changes it made during its active phase have been rolled back. A committed

or aborted transaction is said to be *terminated*.

If a failure occurs, all the information in the database buffer is lost. Before the failure there might have been active transactions executing in the database and all the changed data pages might not have been written to disk, i.e. the database is not in a consistent state. The database system has to guarantee that changes of all committed transactions are durable and all changes made by transactions that were active during the crash are rolled back. The system cannot know if the unterminated transactions already had performed all their operations and so all of them have to be rolled back as a part of *crash recovery*.

Log-based recovery is the most common recovery scheme [SKS97]. During normal processing enough information is written to *log* to recover the modifications made by the transactions. The log records produced by a transaction have to be written to disk prior to committing the transaction. By examining the log and the data on disk the recovery system can restore the database state at the point of the crash. The modifications of the committed transaction found in the log but not on disk are *redone*. The modifications of non-terminated transactions are rolled back and if some of their modifications are found on disk they are *undone*. The modified pages are written to disk and normal processing can continue.

In *physical logging* the *before-image* and the *after-image* are logged. An image contains the location and content of the bytes under modification. Undo or redo can be performed by writing the bytes in before-image or after-image on disk, respectively. In *logical logging* the operation type (insert/delete) and the new and old values of the record are logged. Currently a combination of these schemes, *physiological logging*, is used. In it the logging is physical on the page and logical on the record level. The transaction, page and record identifiers and old and new values are put in the log. The redo operations are always made physiologically on the same page they were done originally, but physiological undo might not always be possible. The added record might have been moved to another page due to insert operations of other transactions. If physiological undo fails, logical undo is performed.

All the transaction's operations are not written to the log, only those that modify the database are logged. Each log record is given a *log sequence number* (LSN) that identifies the log record and place the records in sequential order. Each log record is also attached the LSN of the next log record to be undone in the case of recovery processing. The undo operations (also called *compensation log records, CLRs*) can be redone, but are never undone. Each data page has a *pageLSN* field that points to the last log record that has modified the page. All the active transactions' IDs are kept in *transaction table* and the modified pages' IDs in *modified-pages table*.

The log is stored on disk, but the log records have to be written on the log pages in the buffer. *Write-ahead logging* (WAL) is a method where all the log records whose LSN is greater than or equal to the pageLSN of a certain page are written to disk before the page is written.

Steal policy allows the buffer manager to write a dirty page on the disk, whereas *no-steal policy* forbids this. With no-steal policy the undo operations are never

performed as no dirty data gets on disk. Considering when the clean pages are written, there exist two policies *force* and *no-force*. In force policy, all the pages a transaction has modified are written to disk at commit time. No-force does not require a page to be written on disk while it is still needed. When some pages are frequently modified, force policy performs poorly. In general, database systems use steal and no-force policies [GR93]. The *force-log-at-commit* policy requires the log to be written on disk at commit time even if no-force was used for modified buffer blocks.

In recovery the log has to be studied as far as it has log records whose modifications are not found on the data pages. With the no-force policy it is possible that the whole log since the database was started has to be processed. *Checkpoints* are made to reduce the amount of work needed in the case of a crash. In *complete checkpoint* all the modified pages are written to disk using WAL and a checkpoint log record is written. In crash recovery only log records after the last checkpoint have to be processed. Making a complete checkpoint temporarily suspends normal transaction processing and therefore *fuzzy checkpoints* [SKS97] are widely implemented. A list of modified buffer pages is created and the checkpoint record is written to log. Each page in the modified pages list is written to disk as a background process, normal processing is not suspended. When all the pages are written to disk, the LSN of the checkpoint record is written to disk. This is the last "complete" checkpoint and is the point from which the log has to be scanned in recovery.

ARIES (algorithm for recovery and isolation exploiting semantics) [MHL+92] is a recovery and concurrency control method that supports WAL, record level locking and fuzzy checkpoints. PageLSNs are maintained to keep track of the logged modifications already applied to the pages. Log records are written from every modification even during rollbacks. In restart recovery the log is scanned from last checkpoint to the end of the log. This *analysis pass* gathers information about dirty pages and transactions that were in progress at the time of the checkpoint.

During *redo pass* the state in which the database was at the moment of the crash is reached. The smallest pageLSN from dirty-pages table is set as the *redoLSN* and the modifications from redoLSN to the end of the log are repeated. When processing a log record the corresponding page is latched and its pageLSN is checked. If the update is not on the page, it is repeated as the log record indicates. When the end of the log is reached all the pages are in the same state they were when the crash occurred.

All transactions that were active when the failure occurred are aborted and rolled back as a part of the *undo pass*. The abortion of the transactions that were already rolling back is completed. During normal processing at some intervals and after the restart recovery a fuzzy checkpoint is made. It differs from the fuzzy checkpoint presented above in that when an *end-checkpoint* record is logged, the checkpoint is considered to be complete and the LSN of it is written to disk. The end-checkpoint record might be written on disk before all the dirty pages are, so in recovery the log might be scanned beyond the begin-checkpoint record. The writing of the end

checkpoint can be delayed until all the modified pages have been written from the buffer to the disk.

2.5 Concurrency Control

The purpose of concurrency control is to ensure isolated (as in ACID) execution of each transaction even if there are multiple transactions executing simultaneously. To each transaction the database appears as it was the only transaction executing in it. The transaction's effects are not visible to other transactions until the transaction has committed. Concurrency control makes the database operate more efficiently while creating an illusion of serial execution of the transactions. Concurrency control works in close co-operation with logging and recovery management to provide industrial strength transaction support. The general methods used to accomplish concurrency are locking [Moh90, ML92, Tho98], time stamp and validation based methods [Tho98], and multiversioning [BG83, Tho98]. Each of them are discussed briefly in this section.

The order of *insert* (I), *delete* (D), *write* (W) and *read* (R) operations of transactions is called a schedule. When multiple transactions are run so that each one terminates before another one begins, the schedule is said to be serial. Allowing concurrency means that the operations of various transactions are shuffled. The shuffling has to be done in a manner that does not violate the isolation of transactions. If the shuffled schedule is equivalent to the serial one, it is *serializable*. If the isolation of transactions is not preserved, the consistency of the database might be broken.

The *degrees of isolation* in SQL are *serializable*, *repeatable read*, *read committed* and *read uncommitted*. The default level is serializable. In the literature there exists one more level that resides between serializable and repeatable read. It is *cursor stability* and prevents the *phantom problem*, which appears when one transaction has read a key range and another transaction inserts a tuple inside the range.

General locking methods implement a single writer–multiple reader scheme. Several transactions are allowed to read a data item, assuming no transaction is updating it simultaneously. Respectively, a transaction is allowed to update a data item if no other transaction has locked it. If a transaction requests access to an item and it cannot be granted immediately (due to the preceding rules), a *lock conflict* arises. Conflicting locks are presented in lock compatibility tables [GR93, SKS97]. The transaction asking for a conflicting lock has to wait on the resource. A *wait-for graph* is maintained between the *blockers* and the *waiters*. A cycle in the graph indicates that a *deadlock* exists. Deadlock detection is carried out in order to find the cycles in the graph. This can be done on periodical basis. Deadlocks are solved rolling back or aborting at least one of the transactions involved in the cycle. The aborted transactions might be restarted.

A locking protocol states what locks and for how long are acquired in different situations. Different protocols guarantee different degrees of isolation. A *commit-duration lock* is one that is kept until the transaction commits, locks released earlier

are *short-duration locks*. The primary concurrency control method for centralized databases is *two-phase locking* [SKS97, Tho98]. The two-phase locking protocol is divided to growing and shrinking phases. In the growing phase all the locks needed by the transaction are acquired, but none is released. In the shrinking phase, all the locks acquired in the first phase are released, but no new locks are acquired. However, two-phase locking does not prevent *cascading rollbacks*, where rolling back one transaction necessitates rolling back other transactions as well. To avoid cascading rollbacks the *strict* two phase locking protocol is used [SKS97]. In it, all exclusive locks are held until transaction commit. Another variant is *rigorous* two phase locking in which all locks are held until commit time.

Another locking protocol is *key-value locking* or *key-range locking* [Moh90, Lom93, GR93]. In this scheme locks are acquired and released as follows. While reading x, a commit-duration S-lock is acquired on the key x. The insert of x acquires a commit-duration X-lock on x and a short-duration X-lock on the next key y (the smallest y so that $x < y$). The lock on y is released when the new record is inserted. Deleting x acquires short-duration X-lock on x and a commit-duration X-lock on the next key y. The lock on x is released after it is deleted.

The *next-key locking* used with insert and delete operations prevents phantoms [GR93]. If T_1 has read a key range from x_1 to x_3 (x_2 does not exist) it holds S-locks on the keys. If T_2 now wants to add x_2, it needs an X-lock on x_2 and its successor x_3. T_2 cannot have the lock for x_3 until T_1 has released it, so T_2 has to wait for T_1 to commit. If a transaction wants to abort, no new locks need to be acquired, the undo operations are secured with the commit-duration locks acquired earlier. The SQL isolation degree serializability is guaranteed with key-range locking.

It might be desirable to be able to lock larger parts of the database instead of individual data items, that is, to support multiple levels of *granularity*. The data items are of different size and a hierarchy among them is defined; a database contains relations which contain records. New lock modes are used to implement granularity; *intention shared* (IS), *intention exclusive* (IX) and *shared and intention exclusive* (SIX) locks. The lock compatibility matrix is shown in table 1. To lock a key, first intention locks on higher levels of granularity have to be acquired. For example, to update a key, first the database and the relation have to be locked with an IX-lock, and the key with a X-lock. Locking a data item with a S- or X-lock explicitly locks all the lower level items. When scanning a whole relation, each data item would not have to be locked explicitly if we would lock the whole relation. *Lock escalation* happens when the number of locks on one level of granularity exceeds some predefined value, a lock on the data item on a higher level is acquired. For example in the case of a table scan, when n record level locks are acquired, the system decides to lock the higher level item, the whole relation.

In two phase locking the order of conflicting transaction pairs is determined by the order they have asked for incompatible locks on a resource. An alternative approach is to select an ordering in advance. The most common method for this is the *timestamp-ordering protocol* [SKS97]. Every transaction is given a unique

Table 1: The lock mode compatibility matrix.

	IS	IX	SIX	S	X
IS	x	x	x	x	
IX	x	x			
SIX	x				
S	x			x	
X					

fixed timestamp (TS). Each data item is associated with a *read-timestamp* (RTS) and a *write-timestamp* (WTS). These timestamps mark the transaction execution order. $RTS(x)$ and $WTS(x)$ denote the largest timestamps of transactions who have read and written the item x, respectively. The timestamp-ordering protocol ensures that any conflicting read and write operations are executed in the order the timestamps indicate. Suppose a data item x has been read by a transaction T_j, i.e. $RTS(x) = TS(T_j)$, and its last value is written by a transaction T_i, i.e. $WTS(x) = TS(T_i)$. Now, all transactions T_k whose timestamp TS is newer than or equal to that of T_i can read x, i.e. $TS(T_k) \geq WTS(x)$. All transactions older than $WTS(x)$ trying to read x are rejected. When T_k is trying to write x, it gets rejected if $TS(T_k) < RTS(x)$ or $TS(T_k) < WTS(x)$, otherwise it is allowed to write x. A rejected transaction is rolled back, assigned a new timestamp and restarted.

Validation based or optimistic concurrency control [SKS97, Tho98] is based on the assumption that in most cases the transactions do not update the same data items. Several dirty copies of one data item are given to distinct transactions' private workspaces. The transactions execute in three phases. First, dirty copies are obtained and updated if necessary. Second, the validation phase ensures serializability by checking for conflicting data operations. Arisen conflicts are solved by aborting transactions that fail the validation. The transactions that passed the validation enter the third phase where transactions begin their commit work. They write the log records on disk and the updated data items on the database buffer. While having several suggested implementations and improvements, validation based methods are less suitable for high performance transaction processing [MPL92].

Multiversion concurrency control methods [BG83, AS89, MPL92, SKS97] enhance the methods presented above in that they do not require all the readers to wait or roll back their execution if the data item they are trying to read has been updated after the start of their execution. The concurrency control system maintains the data items' older versions. When a transaction reads an item, it is given the appropriate version of it. Multiversion methods are divided into *multiversion timestamping* and *multiversion locking* [BG83]. In multiversion timestamping all read operations are executed, i.e. none gets rejected. A write operation of a transaction is rejected if a younger transaction has already read the value. Otherwise a write is executed.

Transactions that perform read operations may perform two disk operations instead of one as they need to update the timestamp information [AS89]. Multiversion

locking methods try to reduce this overhead [AS89, MPL92]. Read operations are executed without acquiring any locks, write operations acquire the locks as before. Non-locking reads can be performed without the need to interact with the concurrency control and without the need to update any control information. The read operation in multiversion timestamping had to update the RTS. Common to all multiversion methods is the critical nature of finding the right version for readers. It should cause minimal overhead, if at all. Removing old versions has to be considered as well. Generally old versions can be removed from the system when there are no transactions that could possibly read them. The checking can be done periodically.

The consistency of the physical database has to be preserved as well. This is done with *page latching* [Moh90, SKS97]. Latches are used excessively and, therefore, acquiring and releasing the latches have to be very fast operations. The wait-for graph is not normally maintained on latches. Thus, the latching operations have to be implemented so that deadlocks will never arise. To guarantee that no deadlocks will arise between locks and latches a transaction is not allowed to possess any latches while waiting for a lock. The lock request is said to be *unconditional*, if a transaction starts to wait on a lock that cannot be granted immediately. To prevent deadlocks the transactions ask locks *conditionally*. The lock is granted if it can be granted immediately, otherwise the transaction is informed that the lock is not available. If a lock is not granted, all the latches the transaction holds are released and an unconditional lock request is made. All the other locks the transaction holds are still held. When the lock is granted the transaction reacquires all the latches it released earlier.

Since B^+-trees are the most used access methods, a concurrency control scheme for them is presented [Moh90, ML92]. While inserting and deleting tuples the physical storage structure changes. The changes are performed by *structure modification operations* (SMOs) [Moh96]. A data page is split when it overflows and deleted when the last tuple is removed from it. To execute a SMO, *X-tree latch* is acquired. When a X-tree latch is acquired, all the pages that will be affected by the SMO are in the database buffer. No disk I/Os will be performed during the SMO. The tree is traversed using latch coupling and S-latching the pages. The SMO will start at the leaf level and its effects propagate up the tree as far as needed.

The tree might be in an inconsistent state while a SMO is executing so other transactions executing simultaneously have to be notified about the possible ambiguity of the tree. Each page has a *SM_Bit* (structure modification bit) that is set to 1 as the page gets modified by a SMO. When the SMO completes the bits are reset to 0. The tree latch is acquired only when a transaction traversing the tree sees a page with the SM_bit set to 1 or a SMO is to be performed. A transaction that encounters an active SMO while traversing down the tree will ask for a S-tree latch and therefore be forced to wait for the SMO to complete. Other transactions that do not face SM_Bits set to 1 (because they are traversing in a different part of the tree) will not ask for the tree latch and can execute without disturbance. If a transaction performing a SMO wants to abort, an incomplete SMO will be rolled back. This is possible since other transactions cannot access the pages before the

SMO has completed. After the SMO is completed a *dummy CLR* that points to the last undo-log record before the beginning of the SMO is written to log.

The SMOs are said to be performed as *nested top actions*. A completed SMO will not be rolled back even if the transaction that performed it is rolled back. When an aborting transaction faces a dummy CLR it passes by the SMO leaving its changes intact. The insert operation is done only after the SMO performing a page split has completed. Similarly the deletion is performed and logged prior to removing the page from the index structure. SMOs are serialized using SM_Bits and tree latches. Thus, all operations are performed in a valid state of the tree [Moh96].

2.6 Availability

Distributed databases and the systems to support them have been discussed for a long time [Kim84]. *Shared nothing* [Sto86] was found to be a suitable system structure for distributed database systems. In shared nothing systems each processor has a memory and disk separate of other processors. This is the case today as databases are distributed across separate nodes that communicate via network. Different algorithms were developed to provide concurrent access to the data. Data fragmentation and partition techniques have been widely studied. Data consistency has become a big issue and different schemes have been presented to handle concurrent access to fragmented data maintaining the data consistency across all the nodes [Her87, BGHJ92, LLSG92, BK97]. Distributed database systems are expected to increase the system availability and performance. Replication became the standard technique to distribute a database [LLSG92]. In replication, the database is duplicated across various nodes called *replicas* or *copies*. Each replica contains all the information in the database, but it might be that only one replica is allowed to update the information.

Depending on how the updates propagate to the replicas the replication scheme is called *eager* [KA00] or *lazy* [LLSG92, BKR$^+$99]. In eager replication the update transactions are not allowed to complete before some sufficient number of replicas have been successfully updated. In lazy replication the update transactions commit independent of other replicas. The other replicas are informed about the updates, but their confirmations about committing the updates are not waited for. Eager replication guarantees consistency across the copies but the cost of committing the updating transactions is high. Lazy replication allows many optimizations, as the copies might temporarily be in inconsistent states. Eager replication is also called *2-safe* and lazy *1-safe*. Figure 5 presents the different update propagation schemes.

In *master-copy replication* one replica acts as a *master* and other replicas as *slaves*. The master updates the primary copy of the data and propagates the updates to the secondary copies which reside in the slaves. All the updates happen in the master. Read-only queries can be directed to the slaves thus balancing the load, but are not allowed to perform updates on the data.

In eager master-copy replication the master waits for confirmations from the slaves

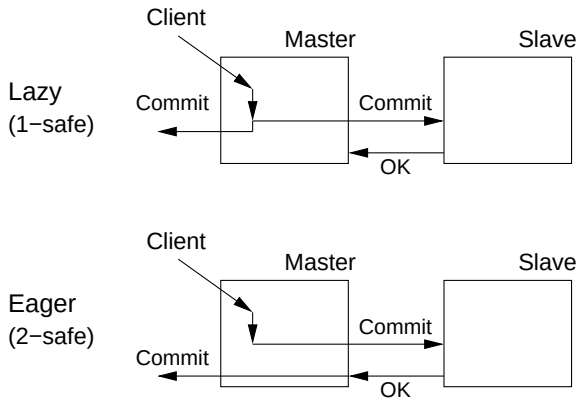

Figure 5: Update propagation schemes in replication.

before committing or aborting. *Two-phase commit protocol* (2PC) [GR93] is used to guarantee the consistency between the copies. 2PC consists of preparation and commit phases. When the master is ready to commit, it asks the slaves to prepare themselves for commit. When the master has received enough answers from the slaves, it makes a decision to commit or abort and tells the slaves to commit or abort as well. If the master does not receive enough answers, it either aborts or commits and logs the update to some location accessible to all replicas. 2PC incurs an overhead to terminating the transactions, but guarantees consistency (to the chosen safety level) between copies. This scheme can be used for a hot-standby backup, where a slave can take the master's role if the primary copy fails.

Lazy master-copy replication allows a response to be sent back to the client before any server coordination has occurred. An update transaction is run in the master and after its commit the changes are propagated to the slaves. The slaves are not guaranteed to be consistent with the master. The updates propagate when the master has already committed or is committing and the slaves might not have executed all the updates in their own copy. As the slaves can serve read-only queries, the clients might see obsolete information.

The master-copy scheme is popular but not the only available replication scheme. *Update-everywhere* allows any item to be updated in any copy, which incurs faster access but complicates the replication protocol [GHOS96, WPS+00]. The eager update-everywhere scheme can be implemented using *distributed locking* or *atomic broadcasting*. In distributed locking the data item to be updated is locked in all nodes prior to its update. If some node does not grant locks on the item, the transaction is delayed and the lock request is repeated after some time. When the locks at all sites are granted the transaction can proceed. The server coordination is done using 2PC and the node that received the client request sends back a response. In the atomic broadcast scheme the client request is broadcasted by the node that received it to all other replicas and each node executes the transaction. No server coordination is done and each node sends a response to the client. The scheme does not suit all situations and weakens the fault tolerance of the system [KA98].

In the lazy update-everywhere scheme the situation is more complicated, because the conflicting transaction may be in progress at various replicas simultaneously. Reconciliation is needed to solve the conflicts and decide the winner transactions. The replicas might be stale and inconsistent. The user might have received a response of a successful transaction while actually the transaction is conflicting with another one and might have to be undone [WPS+00].

In the above master copy schemes, all the data items were supposed to be owned by the same master. It is possible that the ownerships are distributed across different replicas. The update request is sent to the owner of the items to be updated. Timestamps can be used to detect and reconcile update transactions in a lazy update-everywhere scheme [GHOS96].

One of the most important uses of replication is to provide a highly available system. When one database server fails, some other server can instantly take over and continue to provide service to clients. This requires the standby server to be in consistent state with the server in use. Master copy replication is suitable to keep a standby server consistent. The eager protocol guarantees the consistency explicitly. With lazy replication, the consistency has to be compromised or guaranteed with other methods. One such method would be to send dirty pages to the secondary copies as a part of transaction commit in the primary copy and write the log to an external highly available storage. The log is accessible from all the copies and stays accessible even if the primary master fails. When a secondary server becomes the master due to primary master fail, the secondary examines the log and executes all the updates that are not present in its local copy. This way lazy replication can be used to build a highly available consistent system [LLSG92]. When the primary master fails, a secondary copy will take over its tasks. If several standby copies exist, the new master has to be decided among them. The order in which the secondary copies become the master can be predefined by the system administrator, or the replicas can have elections. The winner of the elections becomes the new master.

3 Open Source Database Systems

This chapter presents some of the available open source databases. The design and implementation conventions of the selected databases are presented in the following chapters. Some requirements are set for the databases to be included in the survey. The most important is that the system has to be open source. It has to have commercial support, i.e. at least one company that offers support services and possibly is the developer of the system as well. Tools to build highly available system are required. Transactions and serializability have to be guaranteed and recovery procedures are required. The system must not depend on any single operating system or hardware configuration. Additionally, there must not be any heavy reorganizations or other administrative tasks that disturb normal database operation.

3.1 SAPDB

SAPDB [SAP] is a relational open source database developed by SAP. It offers commercial support as well as other characteristics required except replication. It has a tool called *replication manager* which does not replicate data from one database system to another. Instead it allows data and SQL statements to be loaded from the system into files and correspondingly unloaded from the files into the system.

3.2 SQLite

SQLite [SQL] is an embeddable SQL database engine. It supports a subset of SQL92. SQLite is used via a C library with open database, execute and close database commands. It has no support for replication. Support is provided by the original author of the system with a separate contract. Independent parties have developed drivers and wrappers for different programming languages.

3.3 Firebird

Borland Software Corporation published the source codes of the beta version of their InterBase v6.0 database. A group of developers adopted the code and begun an independent development of Firebird [FB]. Commercial support is available via IBPhoenix [IBP]. However, the general release of Firebird lacks replication. Replication is possible with IBReplicator [IBR], the use of which requires purchasing commercial licenses. The IBReplicator consists of two parts, the *Replication Manager* and the *Replication Server*. The Replication Manager can be run only on a Windows platform while the actual replicated databases can be run on various platforms. Therefore the replication can be used only in a system that includes at least one computer that runs Windows.

3.4 PostgreSQL

PostgreSQL [PG] is a database system offering possibly the best coverage of SQL standards of all open source databases. Concurrency is controlled with multiversioning and therefore it has a *vacuum* operation. The vacuum cleans up old updated and deleted items, updates the relations' statistics used by the query optimizer and guards against transaction identifier wraparound. It is a heavy operation that locks the vacuumed table for the duration of the operation. In the PostgreSQL documentation it is advised to do the vacuum at times when the load is not so high, e.g. night time. This is not acceptable in all cases and heavily used tables need more frequent vacuuming than once a day. It might not be possible to schedule the vacuum appropriately. Replication is supported via various replication development projects. None of them, however, is included in the PostgreSQL releases.

3.5 MySQL

MySQL [MyS] is a relational database that claims to be the world's most popular open source database with its estimated three million installations and 20,000 downloads per day. It has various instances offering commercial support and has also a replication support included in the standard release. MySQL separates the database frontend from the backend, the storage system. Conventionally database systems only support one storage system, whereas MySQL implements a rather unique approach supporting multiple different storage systems called *table handlers*.

Traditional database architecture is shown in Figure 1 (p. 4). MySQL table handler concept is shown in Figure 6. The figure shows only two table handlers, there exist more, but they are not shown here as they do not support transactions. The MySQL frontend offers an interface for the clients, it processes and optimizes the queries and uses a table handler to retrieve the appropriate data. The table handler is responsible for storing and indexing the data, concurrency control, transaction and recovery management. Different table handlers give the application programmers/database administrators the freedom to choose between functionality and optimize between speed and transactional limitations. The default table handler is MyISAM that does not support transactions, but InnoDB and Berkeley DB table handlers provided by Innobase Oy [IDB] and Sleepycat Inc. [BDB] do. The InnoDB and Berkeley DB table handlers are presented in the following chapters.

3.6 Summary

Most of the Open Source databases available depend on some selected operating systems or have heavy maintenance tasks that prevent their use in telecommunications environment. SAPDB, SQLite, Firebird or PostgreSQL do not fulfill the requirements set for the databases to be included in this thesis. They lack replication, are operating system dependent or have heavy administrative tasks. As a result, only MySQL with its transaction supporting table handlers, InnoDB and Berkeley DB, is included in the thesis. InnoDB and Berkeley DB are covered in Chapters 4 and 5, respectively.

4 InnoDB

The InnoDB table handler is studied in accordance with the concepts presented in Chapter 2. The storage management, join calculation, transaction management, concurrency control and replication schemes of InnoDB are presented. Most of the concepts presented in this chapter are InnoDB dependent, but some features are implemented in the MySQL frontend. It is always mentioned if the feature depends on MySQL and not on the table handler and thus directly applies to Berkeley DB table handler as well.

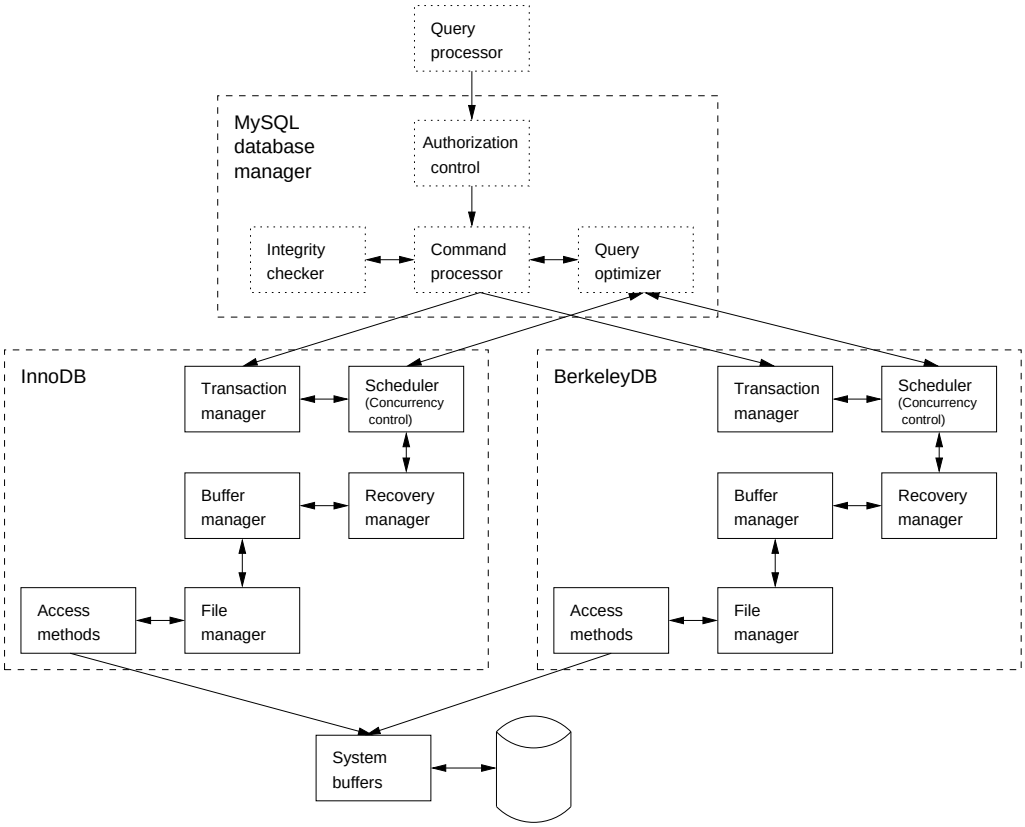

Figure 6: MySQL architecture.

4.1 Data Management

B^+-trees are the only access method supported in InnoDB. The tree is defined as in the previous chapter, the inner nodes do not contain data, and the leaves are connected to both directions, to the predecessor and the successor. The indexes are divided to *clustered* (primary) indexes and *non-clustered* (secondary) indexes. Each table is stored in a clustered index. The clustered index is organized on the primary key of the table. If none exists, a unique row-id is created and used as an identifier. Secondary indexes are created on user-defined attributes.

All the data, except the log, in the database is stored in *tablespace*. The log files are stored separately. The tablespace consists of segments. The segments grow in 1 MB extents as data is inserted in them. Each extent contains 64 pages, 16 kB each. The page size can be changed between 8 kB and 64 kB by recompiling the server source code. The B^+-trees are divided into two segments, one for the *non-leaf index pages* and another for the *leaf pages*. Along with the leaf and non-leaf index page segments a *rollback segment* is maintained. The rollback segment contains the history information of the records serving as undo log and version storage for the multiversioning system. Redo log is stored in separate log files. The storage structure is depicted in Figure 7.

In clustered indexes the leaf pages contain the actual data, the records. The actual data is stored instead of pointers to it to save one disk operation. In the traditional approach leaf pages contain pointers to the data which are then used to access the page containing the data. In InnoDB the index search leads directly to the page containing the data. The leaf pages are stored in sequential order in the leaf index page segment, which makes table scanning more efficient.

The leaf pages of non-clustered indexes contain the primary key (or row-id) value instead of a pointer to the actual record. When the clustered index structure is modified, the secondary indexes stay intact. Accessing a record through a secondary index requires two index searches. First the secondary index is searched based on the attribute value to find the primary key and then the primary index to find the record. If the search on the secondary index returns several primary keys, the clustered index might have to be searched as many times as the first search returned keys. This, however, might not be as bad as it first sounds. From two to three levels of the index structure can be expected to be found in the database buffer, thus not requiring new disk operations [GR93].

There are three different main memory buffers in InnoDB. *Buffer pool* is used to buffer data pages and the rollback segment. *Log buffer* contains the dirty redo log pages. *Additional memory pool* caches information about open table handles and data dictionaries. The relationships between main memory and disk are shown in Figure 8.

The buffer manager observes how index pages in the buffer pool are accessed. The data is stored in B^+-trees, but hash-based access is faster on individual values. Therefore, if a page is accessed frequently enough using the same attribute, the InnoDB buffer manager creates an *adaptive hash index* on the page. The hash index is created in memory and is never written on disk. If the data on the page is updated or the page is not accessed that frequently anymore, the hash index of that page is removed from the memory. Depending on the search pattern found, only the

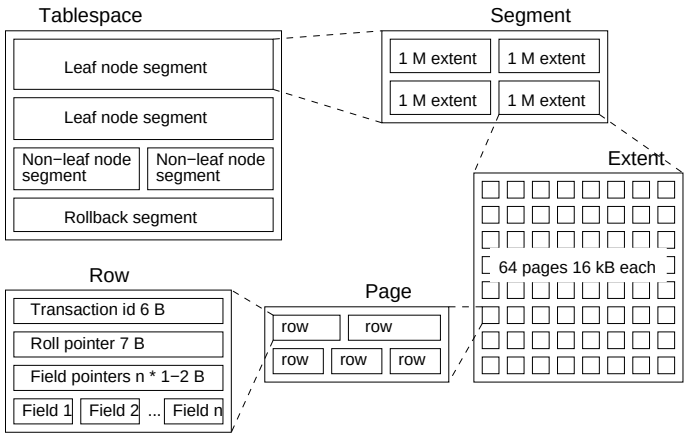

Figure 7: Data file structure [AA02].

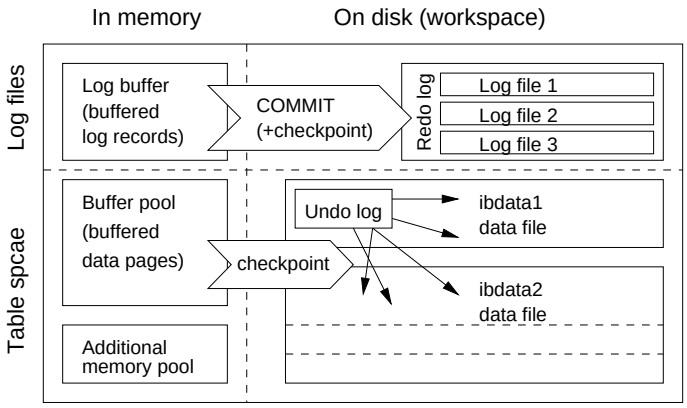

Figure 8: Storage engine [AA02].

prefix of the index attribute can be used to create the hash index.

A special main memory structure called *insert buffer* speeds up insertions. Many times several insertions are done in ascending order of the primary key [Inn02]. Primary indexes handle this case well, as many insertions are done on the same index pages and minimal number of disk I/Os are needed. To secondary indexes the insertions happen in random order and lots of disk operations are needed in order to access all the required index pages. Insertions to primary indexes are always done on appropriate pages in the buffer pool. In the case of secondary indexes, however, the situation is different. If the page on which the new entry is to be inserted resides in the buffer pool, the insertion is done directly to it. If the page is not in the buffer pool, the insertion is done into insert buffer. Periodically the entries in the insert buffer are merged to the secondary indexes. Often several insertions fall on the same index page. All the insertions in the insert buffer that fall on one page are done at the same time, thus the page is retrieved and written only once.

Another structure caches the select queries and their results. If an exact match of the query string is passed to the system again, the result is retrieved directly from the *query cache*. The query is not parsed nor is any data retrieved from the tablespace. When any data is modified, the corresponding entries are flushed from the cache. The query cache is implemented in the core MySQL server and applies to all table handlers.

Doublewrite technology is used to add safety to operating system crashes. The pages to be written to the disk are first written to contiguous tablespace called *doublewrite buffer*. From doublewrite buffer the pages are written to their appropriate locations on disk. The performance gain comes from reducing the need of file synchronization operations. The data can also be stored on *raw disks*. The disk or disk partition contains a single database data file that is accessed by calculating byte offsets from the beginning of the file. This can speed up disk operations as the operating system file cache is not used [GR93]. InnoDB contains two *read-ahead* heuristics; *sequential read-ahead* and *random read-ahead*. When the system notices a sequential access

pattern, it can post in advance read operations on data pages to the I/O system. Then the pages the database system will probably need in the near future are already in the buffer by the time they are needed. Random read-ahead occurs when the system notes that an area, e.g. consecutive pages of a relation, in a tablespace is being completely read. The remaining parts of the tablespace are read to the buffer.

4.2 Join Algorithms

Join processing happens in the MySQL layer of the code. InnoDB or any other table handler is used only to retrieve the corresponding data. The join processing consists of finding a join order for the relations, determining if indexes are used or the relation is sorted and calculating the actual join. The join order is found by approximating the number of rows used per table and the time needed to read the rows and using this data to try different orders. First come relations that have only one row or which are accessed through a unique index or primary key and only few records are retrieved. In general, indexes are used if less than 30 % of the records are retrieved, otherwise a table scan is performed.

The main join algorithm used is a nested loop join. The first step is to determine the join order of the tables. The proper keys to be used are searched, a prefix of a key on multiple columns can be used. The operation on each table is determined, if a table is scanned entirely, if only the index is scanned, or only part of the index is scanned. The tables that contain only one matching row are read first, then the rest of the tables are joined in one sweep. Indexes are used to find single records or to scan a range. Temporary tables are used if an order by or group by statement exists and the sort attributes include attributes from more than one table. The result of the join is put to the temporary table and sorted. The nested loop join is enhanced using indexes where available and appropriate. Even if indexes could be used, a full table scan is usually performed if more than 30 % of the records are retrieved. The optimizer uses statistics to determine whether indexes are used or not.

4.3 Transaction Management

InnoDB is a transactional table handler. It guarantees the ACID-properties with multiversion concurrency control and log based recovery. All SQL statements are executed inside transactions. If the autocommit mode is on, each SQL statement forms its own transaction. Otherwise, transaction borders are set implicitly with *begin* and *commit/rollback* statements. The isolation levels supported are repeatable read and serializable, repeatable read being the default. A transaction sees the changes made by itself or other transactions that committed before the transaction begun its execution. This does not prevent other transactions from updating the records that are read by one transaction.

The readers do not acquire any locks but get the right versions of the records from the multiversioning system. They see the same consistent snapshot of the database

until their commit time, hence the scheme is called *non-locking consistent read* and works as shown in Figure 9. User A sees the same state of the database during the whole transaction even when B inserts a new record to the range A has read. Only when A commits its work and reads the range again, thus advancing its timestamp, it sees the new record. The old versions are maintained in the history list in the rollback segment and fetched as required.

```
                  User A                    User B

              set autocommit=0;
time          BEGIN                     set autocommit=0;
|                                       BEGIN
|             SELECT * FROM t;
|             empty set
|                                       INSERT INTO t VALUES (1, 2);
|             SELECT * FROM t;
v             empty set
                                        COMMIT;
              SELECT * FROM t;
              empty set;

              COMMIT;

              SELECT * FROM t;
              -----------------
              |   1   |   2   |
```

Figure 9: Consistent read.

The consistent read scheme is not always appropriate. If a transaction makes a modification based on a read value, another transaction might update the read value and thus the modification might be incorrect. The previous situation can be prevented making the read operation in *shared lock mode*. In shared lock mode next-key locks are acquired to each index record accessed. Alternatively the transactions can be run with the serializable isolation level, which might reduce concurrency too much.

The log is an undo-redo log, but different from the ones presented in Chapter 2 in that the undo and redo parts are separate. In the ones presented in Chapter 2 both the undo and redo parts were stored in the same log. In InnoDB redo log records are stored in log files and the undo log in the rollback segment in the data file as seen in Figure 8. The rollback segment is used to guarantee the isolation of the read operations and to maintain the undo log records used in crash recovery.

The undo log is divided into two parts, one for inserts and the other for updates and deletes. The insert undo log is discarded after transaction commit. The update and delete undo log records are flushed to disk and moved to the history list at commit. By default InnoDB uses force-log-at-commit policy. It can be set not to use it, as the buffer manager tries to write all modified pages from buffer pool and log buffer

to disk every second anyway.

Fuzzy checkpoints, as described in Chapter 2, are used. The modified pages are written from buffer to disk all the time as a background process. The pages are written in the order of their pageLSNs. The checkpoint LSN is the smallest pageLSN of all the pages in the buffer pool. It is written to the redo log file header to be used by the recovery system. Checkpoints are written when the system is not heavily loaded or when a log file runs out of space. The redo log file is written in circular fashion. When the file is about to fill up, a fuzzy checkpoint is made and the log file is reused from the beginning.

To be able to recover the database correctly the log files have to be archived. The crash recovery is automatic and executes when the server is started after a crash. The recovery method is similar to ARIES [MHL+92]. After analyzing the log, all entries in the redo log after the last checkpoint are executed. Uncommitted transactions are rolled back using the undo log in the rollback segment. After flushing modified pages to the disk the server is ready for normal processing. History entries not needed anymore are removed from the rollback segment. Deleting a record will not remove it physically from the database. When the corresponding undo log records can be discarded, the record and its index entries can be physically removed.

Backing up a database is essential for recovery from hardware crashes. The backups should be taken from both the database and the log files at regular intervals and stored on safe storage. A commercial tool by Innobase Oy is available for taking hot backups of InnoDB. To take a binary backup the server has to be stopped and all data files copied to a safe storage. Dumps of the tables can be taken while the database is running, but to get a consistent snapshot, all the connections should be shut down. Third option would be to use replication. A live backup can be maintained with replication. However, to set up replication, a binary copy of the database is needed.

4.4 Concurrency Control

In InnoDB locking is done on the record level. InnoDB provides concurrency control through a multiversion locking mechanism. Old versions of the records are maintained to show transactions a consistent snapshot of the database at the point of time the transaction begun its execution. At transaction commit time the log buffer is flushed to the disk and the modifications become part of the history list in the rollback segment. When a transaction reads a record and notices that it is newer than its own timestamp, it has to search an old version of the record from the history list. Each record has a roll pointer that points to the version in the history list it replaced.

Consistent read operations acquire no locks, except when the isolation level is set to serializable. In this case the read operation sets shared next-key locks on each record it reads. The same locking scheme is used when a read operation is done at the repeatable read level with *lock in share mode* defined in the query. If an update is

anticipated, the select query can be run with *for update* defined. Exclusive next-key locks are acquired to each record read.

The notion of a next-key lock is different from the one presented in Chapter 2.5 (p. 13). In Chapter 2.5 the next-key lock was acquired on the smallest key after the searched key in the database. In InnoDB a next-key lock locks the record and the gap spanning the record its predecessor. Insert operations acquire only exclusive locks on the records, not on any gaps. E.g. T_1 reads a key range $x_1...x_3$, assume x_2 does not exist. If T_2 now wants to insert x_2, it has to check if some transaction has locked the gap before x_3. T_1 has the gap-lock, if either isolation level is serializable or the query includes *lock in share mode* or *for update* definition. If the range was read as a consistent read at repeatable read level, as is the default, then T_1 does not have any locks and T_2 can acquire the lock on x_2. If, instead T_1 read the acquiring next-key locks to x_1 and x_3 then T_2 cannot insert x_2 as T_1 possesses the lock on the gap.

To prevent phantoms from appearing to the first gap after the locked range, the first record not in the range is also locked. I.e. assume all records whose key is between 50 and 100 ($50 < key < 100$) are retrieved from a table where keys 98 and 100 are present. The last record in the result has 98 as its key. The record 100 is also locked to prevent other transactions from inserting a record with key 99 to the gap between records 98 and 100. In the example the record 100 was locked with a next-key lock that locks the record and the gap before it, even though it is not included in the result, thus preventing other transactions from updating it. Even when next-key locking is not used, phantoms do not appear, because the transactions see the same consistent state up to commit time, as in Figure 9. Using the next-key locking forces serializable execution of the reading and the updating transactions.

Update and delete operations acquire exclusive next-key locks on every record they encounter. A deleted record is marked as deleted, but not removed from the page until the corresponding undo log records have been removed from the history list. There might be other transactions still reading the older versions of the deleted record and they use the deleted record in the page as an access point to the history list. When there are no transactions that might access the old versions, the record and the corresponding history list can be removed.

Lock escalation does not happen as there are no more lock levels. MySQL uses table level locking and those locks can be used with InnoDB type tables also. A table level lock can be granted even if there are conflicting record level locks, because the table level lock is not aware of the record level locks. However, conflicting record locks will not be granted if there is a granted table lock. Granting the conflicting table lock does not endanger the integrity as the record locks maintain it. InnoDB handles lock waits with a waits-for graph and seeks cycles in it to solve deadlocks. The automatic deadlock detector cannot detect deadlocks where table locks are involved, as they are not in the graph. These deadlocks are solved with lock wait timers.

During the tree traversal a tree latch is acquired. Each operation acquires first a S-latch on the tree and searches the tree without separately latching the non-leaf index

pages. They are only bufferfixed. When the leaf level is reached, a latch is acquired on the page and the tree latch is released. Select operations acquire a S-latch while insert and delete operations acquire a X-latch. If a structure modification operation is required the page latch is released and a X-latch is acquired on the tree. The tree is traversed again to the leaf level. The SMO is done first on the leaf and then the update propagates up the tree. During a page split the records are not moved until all necessary pages are split and new pages are allocated. When the SMO is finished, the tree latch is released, the new record is inserted and the records are moved between old and new pages. Deletes never merge pages as the records are only marked as deleted. A purge operation physically removes deleted and obsolete history entries and initiates SMOs if necessary. An adaptive hash search stores the records' page numbers so it has to be updated if records are moved between pages or insertion or deletion occurs.

4.5 Availability

The replication is implemented in the MySQL frontend, thus being an InnoDB independent feature. The replication can be used with InnoDB type tables, though every feature is not available. The replication scheme is lazy master replication. One replica is the master of the whole database, all updates are done via it. Other replicas can serve as backup and take over as the master fails. The slave replicas can also execute read queries to help balance the load of the master.

The master keeps track of all updates made to the database in a log. The slaves read the log and execute the updates in their own copy of the database. Thus the slaves may offer stale data to clients performing read queries, as they might not have executed all the updates the master has written to the log. To set up a replication environment a consistent snapshot of the master's data files has to be made. To make a consistent snapshot the server has to be shut down or all the tables have to be locked. A slave is started with the data snapshot installed. The slave reads the log in the master and catches it up. The master might have been in progress while the slave was not. The slaves keep up with the master by reading the master's log as updates are written to it. The slaves might momentarily be in inconsistent state with the master. The master maintains an index file of the used log files. The index is used to handle log rotation.

A slave can become a master, if a command to do so is executed. However, there is no automatic voting mechanism to provide master takeover. The automatic takeover is planned to be implemented in the near future. An application programmer is responsible for monitoring the master's state and to execute the takeover command if necessary. The new master will start processing the updates and keeping the log. Other replicas will start reading the log from the new master. There is no load balancing technique either. It is in the to-do list as well. The application programmer has to implement a system that directs the read-only queries to different replicas in order to balance the load.

The current state of the replication feature in MySQL does not support high availability well. It is possible to set up a highly available system with MySQL and InnoDB type tables, but the application designer/programmer has to be well aware of the replication techniques. The takeover and load balancing systems have to be implemented entirely in the application end. If a slave loses its connection to the master it will try to reconnect at regular intervals. When the connection is re-established, the slave will synchronize itself with the master. If the slave is down for some time and is restarted, it will again synchronize itself with the master. So restarting any of the replicas causes no problems. The replication support covers crash recovery as well. The redo and undo phases of the recovery produce log records that are executed in the slaves as well.

Emic Networks offers an application cluster [EMI], which provides group commit based fault tolerance and load balancing features for MySQL. The product combines several MySQL databases into an application cluster using replication. With the product a highly available MySQL-based database system could be constructed. As the product is commercial it is not included in this survey.

4.6 Summary

MySQL is a relational database that supports several different storage systems called table handlers. These different table handlers allow the user to choose between performance, required disk space and functionality. The table handlers store the data differently and some support transactions whereas others do not. InnoDB is a table handler for MySQL that supports transactions and foreign key references.

Access methods in InnoDB include only the B^+-tree. The adaptive hash index created in memory when applicable does not cover all the data. The data is stored in the leaf nodes of the B^+-tree and the indexing information in the inner nodes. Concurrency is controlled with multiversioning, where read operations are done as consistent reads without locking. Locking, when needed, happens on record level without the need for lock escalation. Old versions of the records are maintained in rollback segment, which acts as undo log as well. The redo information is stored in log files. Recovery works in the style of ARIES. All the operations are redone and then the unterminated transaction's modifications are undone.

Replication is implemented in the MySQL server and can be used with InnoDB. The replication scheme is lazy master replication, where one server acts as the master and the other replicas have their own copy of the database. The slaves read the update log from the master and execute the modifications in their own database copy. Read operations can be pointed to the slaves, which, however, can temporarily be in inconsistent state with the master. A automatic takeover mechanism is not implemented to change the master in case of primary master failure. Building a highly available system necessitates the application programmer to provide the missing functionality. Manual operation for master takeover is provided by MySQL and can be used in the implementation.

5 Berkeley DB

Berkeley DB [BDB] by Sleepycat Inc. is an embedded database system. It offers application developers/programmers the tools to build highly available transactional recoverable database systems. It is not a database, but a library offering the tools to build one. Sleepycat Inc. offers commercial support for Berkeley DB.

Berkeley DB [Sle, OBS99, SO99] begun as a hashing package for UNIX [SY91]. At the same time a library for transaction support [SO92] was developed. The first release of Berkeley DB in 1991 included the hashing mechanism and a B^+-tree access method. The system evolved and Sleepycat Inc. began to maintain the Berkeley DB and to offer commercial support for it. In 1997 concurrent access was added to the system. Today Berkeley DB is a programmatic toolkit that offers tools to implement concurrent, transactional, highly available database systems.

The toolkit is divided into subsystems; access methods, transactions, memory buffer, locking and logging. Each of these subsystems can be used independently of the distribution. In Figure 10 the subsystems and their interactions are presented. There exist different products that have different characteristics, the Berkeley DB Data Store is an embedded data store without transactions, the Berkeley DB Concurrent Data Store supports concurrent updates, the Berkeley DB Transactional Data Store supports concurrent updates and transactions and finally Berkeley DB High Availability supports transactions and replication. Berkeley DB does not support SQL, nor is Berkeley DB a database server, but one can be built with it.

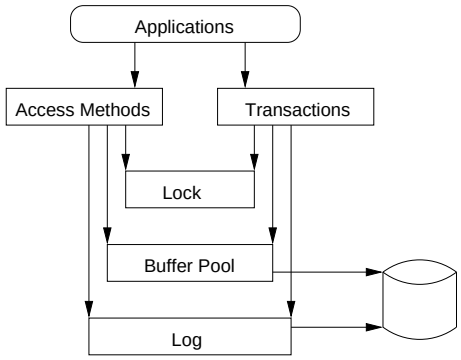

Figure 10: Berkeley DB subsystems.

MySQL has developed an interface that allows one to use Berkeley DB libraries under MySQL frontend as a MySQL table handler. Under MySQL, Berkeley DB's data storage system can be used with SQL. The Berkeley DB library offers tools that are not accessible through the MySQL interface. In the following, the functionality and capabilities of the whole library are explained. There is a mention whenever a Berkeley DB feature is not accessible through the MySQL interface.

5.1 Data Management

Berkeley DB (BDB) offers more access methods than InnoDB where the B^+-tree was the only one. In the BDB B^+-tree, extended linear hashing [Lit80], queue and recno access methods are supported. The recno access method is based on the B^+-tree and is designed to help processing document data [SSL+83].

All access methods store key/value pairs, where keys and values can be of arbitrary length and of any type supported by the used programming language. The types can be for example C structs or user defined objects. However, in recno and queue the key is determined by the system and a further limitation of the queue is that the values are of fixed length.

The recno method can be used to index and access a file that is stored externally, e.g. a text document could be stored based on line numbers. The file data is divided into variable length records by a delimiter. The default delimiter is the ASCII newline character, but any delimiter can be used. If line numbers are the desired record identifier, each separate line represents a record.

The data stored in the B^+-tree and hash table can be of any type supported by the programming language. The records are found based on the keys and comparisons are made on the keys e.g. to solve the order of the records. The hash and comparison functions can be set manually, if the default implementations do not work well with the possibly complex keys. The data pages of the B^+-tree and hash methods are equal. They follow the N-ary storage model page structure idea. The records are stored starting from the end of the page leaving the free area in the middle. In the B^+-tree the data pages are found at the leaf level as in InnoDB. Large records are stored in overflow pages outside the main access method. The BDB table handler, i.e. the MySQL interface to the BDB library functions, supports only the B^+-tree access method.

Each relation resides in its own file. One common buffer pool is shared among any number of threads, each table has its own part in the buffer. The database page size is configurable and determined at the database creation time. Locking happens on page level and changing the page size affects the number of locks acquired when a record is accessed. However, if the page size is set smaller than what is the filesystem block size, the performance might suffer. By default BDB sets the page size to be the same as the filesystem block size.

BDB is not a relational database nor does it support *schemas*. Secondary indexes can be created, but they have to follow certain rules. A secondary index is a separate relation that is created as any relation and associated to an existing relation. The structure of a secondary index is such that the reference key is the relation's key and the referenced value (the key of the primary relation) is the value of the secondary index. The primary keys can be found based on the secondary keys using a join. MySQL does the mapping between SQL and the BDB table handler so the user does not have to be aware of these issues. The case is different when the BDB toolkit is used, then the application developer has to build the indexes manually.

5.2 Join Algorithms

The join algorithm inside Berkeley DB is not used in the table handler. MySQL executes the queries and processes the results using the table handlers only as data retrieval and storage systems. The join algorithms of MySQL were introduced in Chapter 3.2. Here the joining conventions of Berkeley DB are presented, though they are not used in MySQL.

BDB supports natural joins on secondary indexes. All the data is stored as key/value pairs. The primary table has a primary key and a value of some type. The secondary index consists of secondary key/(value of primary key) pairs. The join can be used to retrieve records from the primary table based on a secondary index.

Joining two primary tables is not possible. Several secondary indexes can be used in one join, each has primary keys of the primary table as values. However, secondary indexes cannot be joined on each other, they can only be used to retrieve records from the primary table. To perform a join the application programmer retrieves the desired values from each secondary index involved. The secondary index values are looped through to find all common primary index keys, which are used to retrieve the desired values. The secondary index looping is a modified version of the indexed nested loop.

5.3 Transaction Management

BDB offers tools to implement a transactional recoverable database system. Recoverability is achieved maintaining a log of all the updates and writing it on disk before the corresponding data gets written, i.e. using the WAL protocol as described in Chapter 2.3. The log is physiological and used for both undo and redo operations. By default the log is forced to disk at commit time, as in WAL. The force-log-at-commit protocol can be set off, thus compromising the durability of transactions.

To decrease the amount of work done in the possible recovery phase, checkpoints can be made at regular intervals, either when a certain amount of log files are written or a certain time has passed. During a checkpoint all the modified pages in the memory pool are flushed to disk, a checkpoint record is written to the log and the log is flushed. After a checkpoint is made, the log files can be archived to be used in recovery. Thus the checkpoints are complete. Recovery has to be initiated by the application. A practical way to implement recovery is to run it every time the system is started even though it would not be needed. This way it is not necessary to investigate the different possible states the system might be in. The implementation is simpler and running recovery while not needed does not cause any considerable overhead as it analyzes the system and determines that no further operations are needed.

Different degrees of isolation are supported. Serializability can be guaranteed, but lower levels are achievable also. To increase concurrency the transaction management can be tuned to do dirty reads. The log can be configured not to be forced

and the use of the operation system I/O cache can be avoided. These conventions have to be decided when an application is implemented. The BDB table handler is controlled by MySQL and therefore MySQL makes the decisions. MySQL offers the SQL level begin, commit and abort commands for using transactions. MySQL then passes calls to the BDB table handler.

All the transactional capabilities of the BDB library cannot be used via the MySQL-BDB interface. For example, BDB supports nested transactions, which are not used in the table handler. A transaction can begin *child transactions*. A child transaction can commit or abort independent of its parent transaction. However, if the parent aborts, all its child transactions are aborted and rolled back even if they had committed their work. If a child has not terminated when the parent decides to commit or abort, the child will commit if the parent commits and abort if the parent aborts.

5.4 Concurrency Control

BDB does two-phase locking as presented in Chapter 2. The locks are acquired as needed and released as a part of transaction commit or abort. The locking happens on page level, except in the case of the queue access method, which locks on row level. The lock modes are as presented in the lock mode compatibility matrix in table 1 on page 14. Lock escalation does not happen. The amount of possible lock objects and locks are defined by the application programmer. In addition to the row-level locks the queue access method uses short-duration locks on pages to search the rows. Other locks than those used for data structure traversals are held until transaction commit.

B^+-trees are traversed using lock coupling acquiring S-locks. If a structure modification operation is needed the traversal is done again acquiring X-locks. The recno access method is built on the B^+-tree and thus has the same locking system. However, as the recno maintains the number of records in leaf pages in internal pages, update operations require the internal pages to be X-locked. The locks cannot be released until transaction commit, because the count is not known until the transaction commits or aborts. The presence of numerous update operations decreases the concurrency of the recno access method considerably. In the case of a hash table a metadata page is always locked with a S-lock. X-lock is required only when new pages are allocated to the structure. A waits-for-graph is maintained to detect and solve deadlocks. An automatic deadlock detector can be used to find conflicting waiters. One request is rejected per deadlock. The transaction whose lock request was rejected is rolled back. If the deadlock is not solved another request is rejected and the corresponding transaction is rolled back.

5.5 Availability

MySQL supports increasing availability through replication. The MySQL replication scheme presented in Chapter 3.5 can be used with BDB type tables as well. BDB offers its own replication scheme that can be used in systems developed with the toolkit. Here, the replication scheme of BDB is described, even though it cannot be used with MySQL. BDB supports "single master, multiple slaves" replication. The replication can be built either as eager or lazy.

The application has to provide many critical parts of the replication for it to work as expected. Communications infrastructure, replica priorities, monitoring the replicas' states, security policies and naming are all the responsibility of the application. The application programmer has to be well aware of the different conventions of replication systems and their effects. In the scheme all updates take place in the master, the slaves can serve as read-only servers. The database cannot be partitioned, each replica has a copy of the whole database. The replication can be used to maintain a backup copy of the log. The log-only slave cannot serve read queries, but other slaves might exist in the system to balance the query load.

A slave can become a master if the master or the connection to it is lost. The application has to monitor the state of the master and call for an *election* if it suspects that the master is not functional anymore or it does not know who is the master. In an election the master announces that it is still the master, or a new master is chosen if the old master no longer replies. The application has to name each replica and provide the communication links between the replicas (environments). When an election is initiated, the environment with the highest priority, which is set by the application, becomes the new master. At least $n/2 + 1$ of the environments are required to participate in the election, or the system is no longer functional. The application possibly has to reconfigure the environment when it changes from slave to master and redirect the updates to the new master. The data can be guaranteed to be up-to-date in each replica, thus eager replication can be implemented.

Distributed transactions are supported by the two-phase commit protocol. Its use is not enforced, so a lazy replication scheme can be constructed as well. It is up to the application to decide the replication scheme. The application has to monitor the transaction states and manage global transaction IDs. Recovery is supported on each individual environment, but a distributed transaction's recovery has to be implemented in the application.

5.6 Summary

The basic features of Berkeley DB are a lot like those presented in Chapter 2. The data model is not relational, BDB stores key/value pairs, which can be of any type supported by the programming language used. Indexed nested loop join is supported between a relation and secondary indices on the relation. Recovery is log based. The log is physiological and written using write-ahead logging. Recovery redoes the mod-

ifications found in the log but not in the database and undoes the non-terminated transactions' modifications. Locking follows the two-phase locking protocol on page-level, except with the queue access method that locks on record level. Lock escalation does not happen even though intention locks are used. Replication is supported and can be implemented as lazy or eager replication. To implement eager replication a two-phase commit protocol is used.

BDB does not support SQL nor is it a database server. It is a library that offers application developers the tools to implement data management in their applications. It consists of separate subsystems for logging and locking as well as buffer and transaction management. The subsystems are independent and can be used separately in any application. The BDB library can be used as a MySQL table handler via an interface included in MySQL. All the BDB tools are not usable under MySQL, for example the replication is MySQL-dependent and does not use the BDB replication.

6 Experimental Evaluation

The performance of a database system can be evaluated with benchmarks. There exist generic benchmarks that can be used for this purpose. Probably the most popular and widely accepted is TPC-C on-line transaction processing benchmark [TPC]. Usually a benchmark is run on several systems. The performance and cost of each system is measured and recorded. The performance metric typically is transactions per second (tps) and the price five year cost-of-ownership [Gra93]. From these metrics price/performance figures can be calculated.

The generic benchmarks suit evaluating the overall performance of different systems in database use well. Different database systems can be compared with the same hardware configuration or different hardware configurations can be evaluated using the same database system. Generic benchmarks can be used to give a rough estimate of the system's performance [Gra93]. However, the performance of a specific system depends on many aspects that are not considered in the generic benchmarks. The system should always be evaluated with a benchmark that is application domain specific. A domain-specific benchmark should be relevant, portable, scalable, and simple:

- relevant in the sense that it reflects the application domain,
- portable so that it can easily be implemented on several systems,
- it should apply to small and large computer systems, and
- be simple so that it is understandable.

Mobile telecommunications is an established and still evolving business area that widely uses databases and includes a large amount of data processing. The open

source database systems covered in this thesis will be evaluated with a network database benchmark [Tik02] that mimics a home location register (HLR) [ETS], which is used in mobile networks to store information about the users of the network. The network operators use HLR databases to store subscriber data, location data, network access data, and data about network services, for example call forwardings.

To simplify the benchmark it does not contain all the operations of a HLR. It measures the performance of a DB running the most essential parts of an operational HLR system. Adding a new subscriber or deleting an existing one are not included as they constitute only a very small part of the load. Long transactions and large joins are also left out as they are typically run on administrative databases.

The characteristics of a mobile phone network place strict requirements on network database systems. High and stable performance, scalability, and high availability are typical features of such databases. In this thesis a benchmark is run on selected open source database systems to see how they perform in the HLR type of use. The tests are restricted to only a single server system and concentrate on the performance and scalability metrics of it. Multi-node scalability or availability tests are not included as the replication of the open source database systems to be tested is not 2-safe (see Figure 5 on page 17). 2-safe replication would guarantee no loss of updates.

6.1 The Testing Environment and the Benchmark

The database consists of four tables: Subscriber, AccessInfo, SpecialFacility, and CallForwarding. The basic data, such as the location data, of all subscribers using the network is found in the Subscriber table. The AccessInfo table holds information about network access data, e.g. granting a subscriber access to the 900 MHz and 1800 MHz networks. Network services accessible to a subscriber are stored in the SpecialFacility table. Each of those services might have a number of call forwardings, which are recorded in the CallForwarding table. The number of call forwardings depends, for example, on the state of the subscriber's terminal. The entity-relationship diagram of the database is shown in Figure 11. The CREATE TABLE-definitions for the tables can be found in Appendix 1.

The distribution is uniform within the cardinality range is equal, e.g. 25% of the rows in the Subscriber table have one row in the AccessInfo table, 25% have two rows, 25% three rows and 25% four rows. Three different populations are used to test the databases in situations, where:

1. all the data in the database fits in the main memory (100,000 subscribers),
2. half of the data fits in the main memory (200,000 subscribers), and
3. about 20% of the data fits in the main memory (500,000 subscribers).

When all the values in the cardinality range have an equal probability and the number of rows in the Subscriber table is N, the number of rows in AccessInfo and SpecialFacility tables is $2.5 * N$ and in the CallForwarding table $3.75 * N$.

The data is populated using a client that creates random data according to the cardinality distribution. The attribute values are evenly distributed where appropriate, e.g. the length of a time interval in a call forwarding is evenly distributed between 1 to 8 hours in the CallForwarding table.

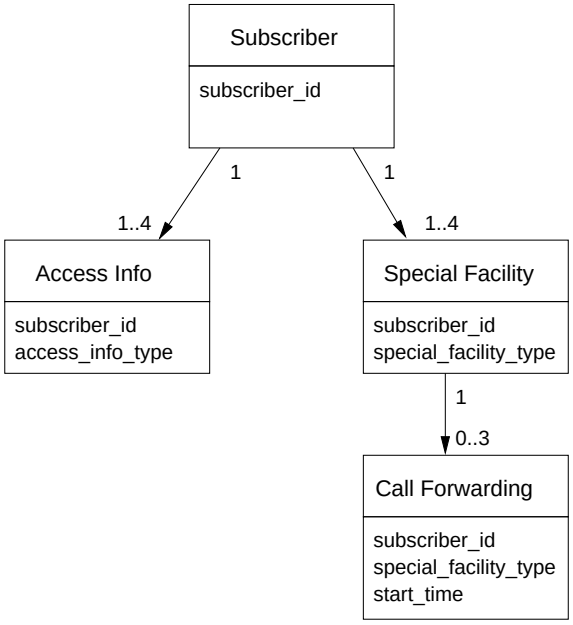

Figure 11: Entity relationship diagram of the HLR database and primary keys of the tables.

There are seven different transactions included in the tests. The SQL clauses for the transactions are found in Appendix 2. Three of the transactions; Get-Subscriber-Data, Get-New-Destination and Get-Access-Data, are read-only transactions, whereas Update-Subscriber-Data, Update-Location, Insert-Call-Forwarding and Delete-Call-Forwarding are updating transactions that change the data in the database. The transactions execute the following tasks:

- Get-Subscriber-Data retrieves one row from the Subscriber table using a subscriber id, which is the primary key of the table.
- Get-New-Destination retrieves a new call destination for a call. It joins the SpecialFacility and CallForwarding tables and retrieves one row (if it exists and the call forwarding is active) with a given subscriber id and a special facility type. These two attributes form the primary key of the SpecialFacility table. According to the cardinality distribution rules approximately 23.9% of all the executed Get-New-Destination transactions return a row.
- Get-Access-Data retrieves the subscriber access data from the AccessInfo table using a subscriber id and a access info type, which is the primary key of the table.

- Update-Subscriber-Data updates a value in both Subscriber and SpecialFacility tables using a subscriber id.
- Update-Location updates the location information of one subscriber in the Subscriber table using a subscriber number, which is a unique key (and therefore has an index) in the Subscriber table.
- Insert-Call-Forwarding inserts a row in the CallForwarding table. The insert is possible if a row with the given subscriber number and start time does not yet exist. According to the cardinality and attribute value distribution rules 50% of the insertions are successful.
- Delete-Call-Forwarding deletes a row from the CallForwarding table according to a subscriber number. According to the cardinality and attribute value distribution rules 50% of the deletions are successful.

The database resides in one computer which acts as the server. A multi-threaded client program executes in another computer. The life cycle of a client thread is shown in Figure 12. The transaction type to be executed next is randomly selected according to the transaction mix shown in Table 2. Each client thread simulates a separate client. Each thread has its own connection to the database and executes simultaneously, but independently of the others. The response time target is that 90% of the read transactions finish within 10 milliseconds when the population is 100,000 subscribers and within 50 milliseconds when the population is 200,000 or 500,000 subscribers. Increasing the number of threads in the client increases the load in the server and therefore makes the individual transaction response time longer.

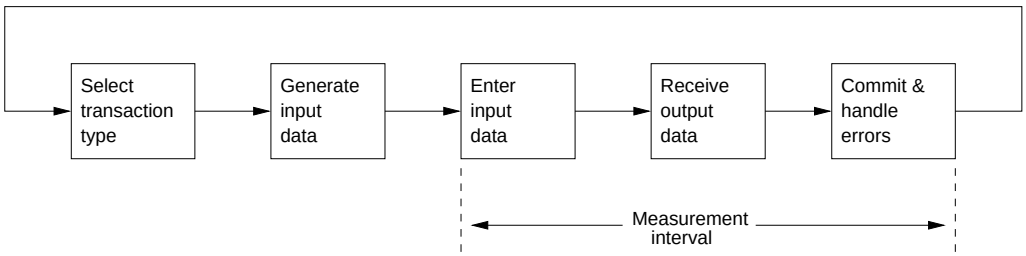

Figure 12: The cycle executed by a test client.

The performance metrics are the maximum qualified throughput (MQTh) and the durations of the individual transactions. MQTh is the number of all completed transactions per second. The test is conducted in a steady state which is reached by running the test client 10 (for 100,000 subscribers) or 30 (for 200,000 and 500,000 subscribers) minutes prior to the beginning of the measurement interval. The test interval is 120 minutes. Each completed transaction is individually timed (see Figure 12). Response times are measured in the test client. The test client uses open database connectivity (ODBC) [ODB] to connect to the database server.

Table 2: The transaction mix distribution.

Transaction type	% of mix
Get-Subscriber-Data	35
Get-New-Destination	10
Get-Access-Data	35
Update-Subscriber-Data	2
Update-Location	14
Insert-Call-Forwarding	2
Delete-Call-Forwarding	2

The server computer is a 1,000 MHz Pentium3 with 256 MB of main memory. The client computer is a 266 MHz Pentium2 computer with 128 MB of main memory. Both computers run the Linux operating system: the server has kernel version 2.4.18-6 and the client version 2.2.18-4. The computers were connected with a 100 MB Ethernet. The tested databases are *MySQL 4.03 max*, which includes both InnoDB and Berkeley DB (BDB) table handlers, and a commercial database. In the following discussion the commercial database is abbreviated as *CDB*.

The ODBC library used for MySQL was *MyODBC 3.51.04*. The commercial database had its own ODBC library which came with the distribution. The transactions were executed as prepared statements over the ODBC connection. Of the database configuration parameters only the memory size-related parameters were changed. The most important parameter was the cache size. The biggest possible value for the cache size was searched by trying different cache sizes so that the server computer did not begin to swap between main memory and the disk. Of the available 256 MB main memory, approximately 170 MB was given to the database.

The test was run with three different populations and using three different databases resulting in nine tests in total. The SQL isolation level was repeatable read. The isolation levels were explained in section 2.5 on page 12. The force-log-at-commit (see section 2.4 on page 11) policy was used.

The number of checkpoints made during the evaluation period and the checkpoint interval is reported, as well. Berkeley DB library offers the possibility to affect the checkpoint interval, but the MySQL interface to the library does not, so the Berkeley DB checkpoint interval cannot be changed. CDB's documentation states that it uses a checkpoint interval which most probably is the best trade-off between temporary performance loss during the creation of a checkpoint and advantage in recovery situation. This default checkpoint interval of CDB was not changed.

6.2 Database in Main Memory

The numerical summary of the test with the population of 100,000 subscribers is shown in Table 3. The load was increased by changing the number of threads in

Table 3: Numerical summary of the test with 100,000 subscribers.

	InnoDB	BDB	CDB
MQTh (tps)	1,134.6	976.0	537.0
Number of threads	5	4	2
Number of checkpoints during evaluation period	N/A	20	40
Checkpoint interval (min.)	N/A	6.0	3.0
Number of transactions completed during evaluation period	8,168,836	7,027,369	3,866,407

the test client. The limiting factor was the duration of the read transactions of which 90% must complete within 10 milliseconds. So the number of the threads was increased until the 10 ms limit could not be kept. The number was chosen separately for each database. The read transaction performance is crucial as they constitute 80% of the overall load. Table 4 shows the 90^{th} percentile and average response times of each transaction type.

The largest part of the individual transactions last only a few milliseconds. As an example of how the transaction response times distribute, Figure 13 shows the distribution for the Get-Subscriber-Data transaction with the population of 100,000 subscribers. All the response time distributions are not shown as the form of the graphs is very close to the one presented in Figure 13.

The graphs in Figure 14 show the change of the MQTh value during the ramp-up and evaluation period with the population of 100,000 subscribers. The effect of checkpointing is obvious. InnoDB does fuzzy checkpointing and therefore has very stable performance throughout the test. Berkeley DB suffers from checkpointing significantly. The overall performance is close to that of InnoDB, but during checkpointing the performance degrades about 25%. The overall performance of CDB is a lot poorer and the checkpointing degrades the performance even further.

Table 4: Transaction durations (90^{th} percentile / average) in milliseconds with 100,000 subscribers.

	InnoDB	BDB	CDB
Get-Subscriber-Data	9 / 4.8	10 / 3.7	5 / 3.0
Get-New-Destination	5 / 2.9	7 / 2.6	9 / 6.5
Get-Access-Data	4 / 2.4	7 / 2.4	3 / 2.0
Update-Subscriber-Data	11 / 5.0	18 / 6.8	8 / 4.8
Update-Location	8 / 3.8	12 / 4.4	5 / 3.3
Insert-Call-Forwarding	9 / 4.6	17 / 5.8	7 / 4.0
Delete-Call-Forwarding	9 / 4.6	17 / 6.1	7 / 4.0

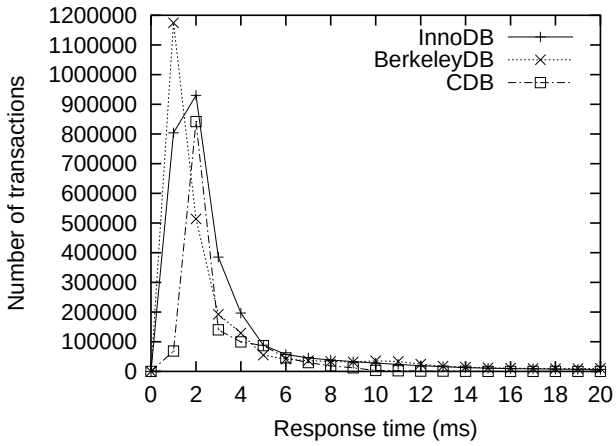

Figure 13: The response time distribution for the Get-Subscriber-Data transaction with 100,000 subscribers.

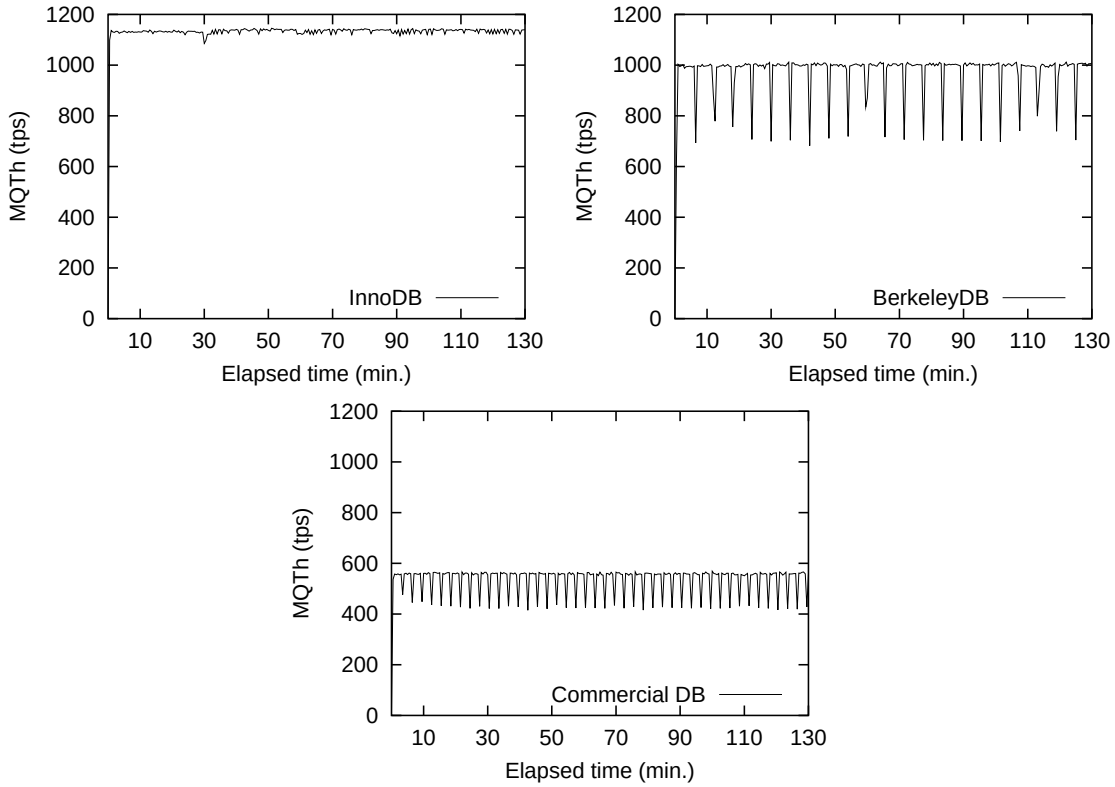

Figure 14: MQTh versus elapsed time, population of 100,000 subscribers.

With the population of 100,000 subscribers InnoDB had the best overall performance. The number of checkpoints of InnoDB cannot be reported as it implements fuzzy checkpointing as introduced in Chapter 4.3 (p. 25). With InnoDB and Berke-

Table 5: Numerical summary of the test with 200,000 subscribers.

	InnoDB	BDB	CDB
MQTh (trans./sec.)	1,013.0	1,048.5	577.9
Number of threads	20	20	10
Number of checkpoints during evaluation period	N/A	22	40
Checkpoint interval (min.)	N/A	5.5	3.0
Number of transactions completed during evaluation period	7,298,663	7,549,360	4,160,732

ley DB the slowest read transaction is Get-Subscriber-Data whereas with CDB it is Get-New-Destination. As all the data fits in the database cache this is due to the transaction profiles. The Get-Subscriber-Data transaction retrieves data from one table only whereas the Get-New-Destination transaction retrieves data from two tables.

6.3 Half of the Database in Main Memory

The 90^{th} percentile constraint for read transactions with 200,000 subscribers was 50 milliseconds. With 200,000 subscribers the overall performance of Berkeley DB was slightly better than that of InnoDB (Table 5). Again, the overall performance of CDB reached only half of that of InnoDB and BDB. The number of threads was a lot bigger than during the tests with 100,000 subscribers. This was due to the increased 90^{th} percentile constraint. However, with CDB the number of threads was not increased above 10, because it did not further increase the throughput, but just slowed down the individual transactions (Table 6). With 10 threads the 90^{th} percentile of the slowest read transaction was 37 milliseconds. The transaction type-specific metrics are shown in Table 6.

The performance graphs in Figure 15 show the MQTh value changes during the

Table 6: Transaction durations (90^{th} percentile / average) in milliseconds with 200,000 subscribers.

	InnoDB	BDB	CDB
Get-Subscriber-Data	47 / 19.2	46 / 20.0	28 / 15.5
Get-New-Destination	29 / 15.0	31 / 13.7	37 / 24.5
Get-Access-Data	20 / 12.8	31 / 13.5	24 / 13.3
Update-Subscriber-Data	83 / 33.0	80 / 31.2	41 / 26.0
Update-Location	62 / 27.3	53 / 21.3	29 / 18.0
Insert-Call-Forwarding	76 / 28.8	72 / 27.7	38 / 23.8
Delete-Call-Forwarding	72 / 27.7	72 / 28.5	37 / 22.1

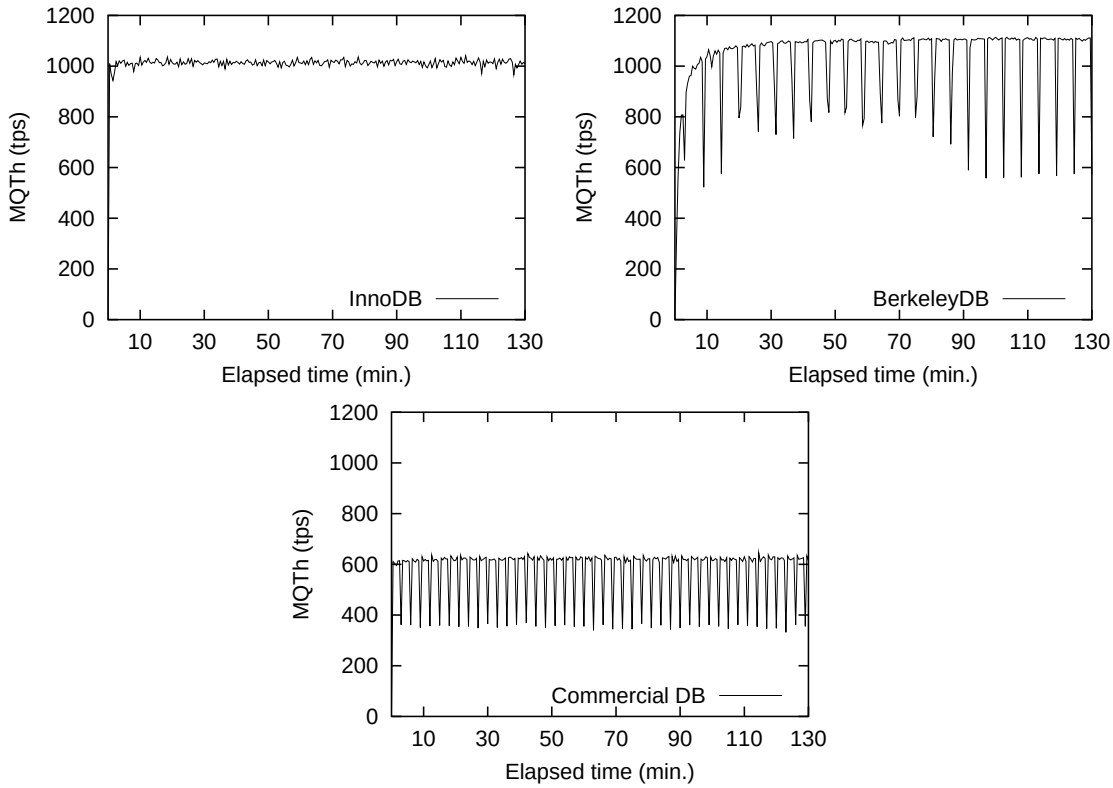

Figure 15: MQTh versus elapsed time, population of 200,000 subscribers.

ramp-up and evaluation periods with 200,000 subscribers. Compared to the tests with 100,000 subscribers the performance of InnoDB degraded slightly as Berkeley DB and CDB performed a bit better than with the smaller population. Due to the relaxed 90^{th} percentile limit the overall performance did not change much while the individual transactions' response times increased. The MQTh value of Berkeley DB degrades 50% during checkpointing, as does that of CDB. InnoDB on the other hand continues to perform as stably as with the smaller population.

6.4 Database Mostly on Disk

The overall performance of all databases degraded significantly as the population was increased to 500,000 subscribers, as shown in Table 7. Only about 20% of the data in the database fits in the memory, so the buffer management becomes very important. The database operation with this large population was almost entirely disk-bounded.

The individual transaction response times of InnoDB and BDB were very close to each other with the smaller populations. With the population of 500,000 subscribers the Get-New-Destination transaction was considerably slower than the other read transactions with InnoDB (Table 8). It can be seen from the transactions' SQL-

Table 7: Numerical summary of the test with 500,000 subscribers.

	InnoDB	BDB	CDB
MQTh (trans./sec.)	357.8	604.7	354.9
Number of threads	3	20	5
Number of checkpoints during evaluation period	N/A	13	27
Checkpoint interval (min.)	N/A	9.0	4.5
Number of transactions completed during evaluation period	2,576,259	4,353,906	2,555,454

clauses (in Appendix 2) and the transaction mix (Table 2) that only 16% of the transactions handle data in the SpecialFacility table. With the largest population, where only about 20% of all the data fitted in the main memory, the data in the SpecialFacility table is not found in the cache. So this Get-New-Destination transaction is considerably slower than the other read transactions, which can be seen from the transaction type response times of InnoDB in Table 8.

However, Berkeley DB does not produce the same result. This leads to the conclusion that the caching differs in the products. With BDB all the read transactions lasted almost the same time. This affected the amount of threads that could be used to create the load. With InnoDB only 3 threads could be run whereas with BDB, 20 threads were running. This was due to the slow execution of Get-New-Destination transaction in InnoDB.

Figure 16 shows the performance graphs with the largest population. All the databases perform significantly more poorly compared to the performance with the smaller populations. InnoDB has suffered the most, Berkeley DB's and CDB's performance is about half of the performance they reached before. Because of checkpointing the performance of Berkeley DB degrades about 90% during checkpointing and more than 50% with CDB. InnoDB continues to perform as stably as before.

Table 8: Transaction durations (90^{th} percentile / average) in milliseconds with 500,000 subscribers.

	InnoDB	BDB	CDB
Get-Subscriber-Data	5 / 3.3	41 / 26.6	19 / 8.8
Get-New-Destination	43 / 13.5	43 / 27.2	41 / 21.5
Get-Access-Data	16 / 3.8	39 / 24.6	18 / 7.6
Update-Subscriber-Data	48 / 23.3	136 / 80.8	61 / 35.2
Update-Location	36 / 18.4	85 / 52.2	48 / 26.2
Insert-Call-Forwarding	58 / 22.8	115 / 69.3	59 / 32.7
Delete-Call-Forwarding	43 / 17.9	113 / 67.7	48 / 25.8

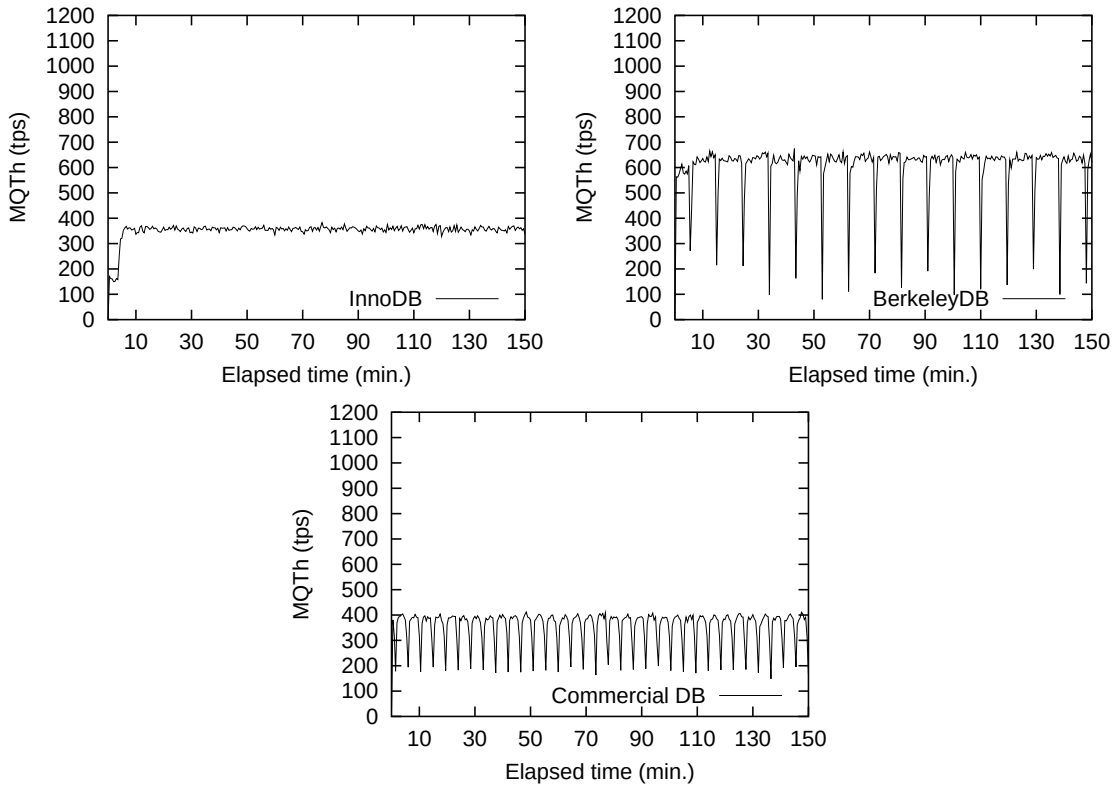

Figure 16: MQTh versus elapsed time, population of 500,000 subscribers.

InnoDB's performance degradation is most probably due to the buffer management. The slowest read transaction of InnoDB is different than with the smaller populations. The calculation of the slowest read transaction's result involves joining rows from two relations, which are used only in a small part of the transactions. Therefore, as only about 20% of the data fits in the cache, these rows most probably have to be retrieved from the disk to the buffer.

6.5 Effects of the Size

The MQTh value of InnoDB and Berkeley DB is almost equal with the two smaller populations. The commercial DB reaches to half of the performance of the other databases with the smaller populations. With the largest population InnoDB falls to the level of CDB, but continues to perform stably. Figure 17 concludes the MQTh values gained in the tests and the effect of the database size.

The overall performance itself is not a sufficient metric as checkpointing affects it. The effects of checkpointing can be seen in the MQTh graphs in Figures 14, 15 and 16. The cost of checkpointing is higher as the database size grows. Berkeley DB's performance degrades about 25% with the smallest population, but up to 90% with the largest. Berkeley DB has clearly the best overall performance accordgin

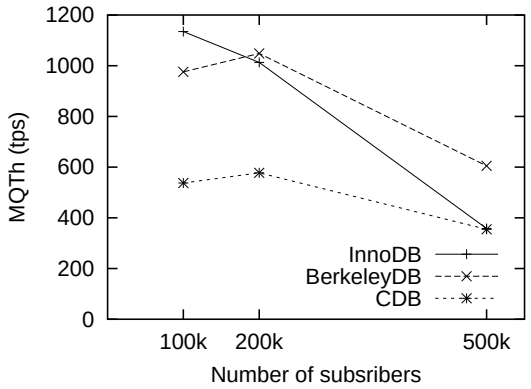

Figure 17: MQTh values of all database products and database populations.

to the MQTH values with the largest population, but the cost of checkpointing is extremely high. Almost no transactions are able to execute when a checkpoint is made.

It could be argued that the overall performance of Berkeley DB and CDB would be stabler if the checkpoints were made more frequently. More frequent checkpoints would result in stabler performance, but degrade the performace as well. With frequent checkpoints using the largest population the MQTh value of Berkeley DB is expected to drop to the level of InnoDB or below, possibly close to 250 tps. However, as the MySQL interface to the Berkeley DB library does not offer the possibility to change the checkpoint interval, the assumption cannot be verified.

The response time distributions maintained their form while the database size grew. Figure 18 shows the distribution of the individual Get-New-Destination transactions' durations for all the populations. With CDB this was the slowest of the read transactions throughout the evaluation. While with InnoDB and Berkeley DB the largest part of the transaction responded in a few milliseconds, with CDB the response came several milliseconds later.

With the largest population InnoDB's slowest read transaction is different than with the smaller populations. Due to the slowness of one of the three read transactions the number of threads had to be reduced to 3 which yields to rather moderate MQTh value. InnoDB's cache seems to work differently when only a small part of the data fits in the main memory as the slowest read transaction changes. Berkeley DB's slowest read transaction is the same as with InnoDB; Get-Subscriber-Data with the two smaller populations and Get-New-Destination with the largest population. With Berkeley DB the number of threads can be kept the same as with 200,000 subscribers; 20 threads are used. Therefore with the largest population the overall performance is better with Berkeley DB than with InnoDB. With CDB the slowest read transaction is Get-New-Destination with all three populations. CDB cannot handle the same kind of load as InnoDB and Berkeley DB and allows only a small number of threads to be used. Only with the largest population the number of threads is bigger with CDB than with InnoDB.

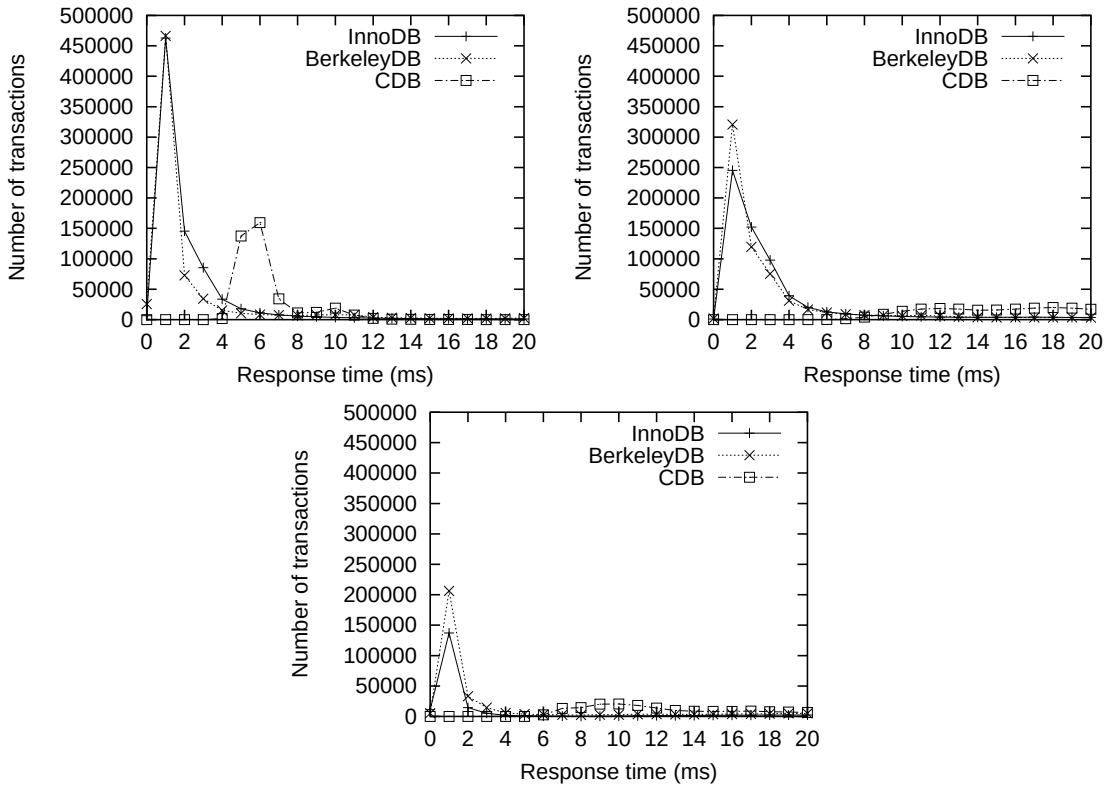

Figure 18: Response time distribution for the Get-New-Destination transaction.

6.6 Conclusions

Checkpointing affects the performance of Berkeley DB and CDB drastically. InnoDB does fuzzy checkpointing, which leads to a very stable performance with all populations. The effect can be seen from the performance graphs. The graphs show the number of executed transactions per second, measured in 30 second periods. With Berkeley DB the performance degrades up to 90% and with CDB up to 50% during checkpointing. To stabilize the performance and weaken the effect of checkpointing the checkpoints could be made more frequently. The number of pages to be flushed to the disk would be smaller, resulting in faster checkpoints. The overall performance would degrade as well, but would result in stabler execution. In a HLR type of use stability is essential. It cannot be allowed that every 5 to 10 minutes the database almost stops responding due to a checkpoint being made.

The checkpoint interval of Berkeley DB could not be changed with Berkeley DB and therefore the presumption above could not be verified. CDB's checkpoint interval could have been changed, but as the overall performance is worse than with the other databases, the effort was not made to stabilize the performance.

The transaction type response time stability and overall performance of Berkeley DB would make it a tempting database for HLR use, but its checkpointing affects its

performance intolerably. The Berkeley DB library offers tools that possibly would help to stabilize the performance, but the MySQL interface to Berkeley DB is not mature enough and does not offer these tools to the MySQL users. CDB's overall performance is rather poor and it suffers from expensive checkpointing as well. Both InnoDB and Berkeley DB outperform CDB. InnoDB performs as well or even better as Berkeley DB with the two smaller populations, but a bit surprisingly falls to the level of CDB with the largest population. InnoDB does not suffer from expensive checkpoints and offers a very stable performance throughout the tests. Regarding to its performance, InnoDB could well be considered for HLR type of use.

It should be stressed though, that the tests here cover only a single server environment. An operational HLR must be highly available. This goal cannot be achieved by a single server, but requires 2-safe replication. MySQL replication is not 2-safe and therefore it cannot be used in real-world telecommunication applications, where the data is required to be consistent in all situations.

7 Summary

The software business is evolving as traditional closed systems are challenged by open source systems, which can be used free of charge in non-commercial systems. Beside operating systems, database systems are a highly developed area of open source software. Long developed and feature-rich open source databases exist on the Internet.

Mobile phones and the networks required for their use involve a great deal of data processing. Embedded databases have been traditionally used in this industry, but now with third generation mobile phone networks and open source databases this situation could be changing. Some requirements are placed on the potential databases that might be considered for mobile telecommunications. The databases are required to have commercial support, they have to offer tools to build a highly available system, they must not depend on any single hardware or software configuration and they cannot have any heavy administrative tasks that disturb normal database operation.

A typical database management system includes various components which are involved in query processing, transaction management, concurrency control, data management, etc. Some chosen topics were covered in order to understand the basics of data and transaction management and availability in database systems. The features were needed to discuss the database concepts used in open source databases.

The open source market was surveyed and potential databases were selected for further studying and experimenting. Most of the available open source databases lack features required in mobile telecommunications systems or depend on some single operating system. These databases were not studied in depth. Only MySQL fulfilled the requirements and was selected for further study.

MySQL differs from traditional databases in that it has various storage systems

called table handlers. Each table handler has its own characteristics and implements its data and transaction management independently. InnoDB and Berkeley DB table handlers were selected for studying and experimenting. The concepts used in the table handlers were rather similar to the basic database concepts. MySQL has replication, but it is not 2-safe thus the replicas can be in inconsistent state compared to the master server. This is not acceptable in critical applications in the mobile telecommunications industry.

Experiments were done with MySQL and its InnoDB and Berkeley DB table handlers as well as with a commercial database to compare the results. The experiments were done running a benchmark in the target databases. The benchmark was based on a home location register. A HLR database is used in mobile phone networks to store data about users of the network. The benchmark consisted of different read and write transactions typical to a functional HLR.

The experiments show that the InnoDB table handler and the Berkeley DB table handler have satisfying overall performance, whereas the commercial database has rather moderate performance characteristics. Berkeley DB suffers greatly from its checkpointing scheme and its performance degrades periodically up to 90%. InnoDB implements fuzzy checkpointing and has very stable performance with all the database sizes tested. However, when the database resides mostly on disk, the performance of InnoDB falls behind that of Berkeley DB.

Because of its checkpointing scheme Berkeley DB cannot be considered to be used in critical mobile telecommunications applications. InnoDB shows very stable performance characteristics and could be an interesting option. However, MySQL does not offer 2-safe replication, which is essential for applications in this target domain. Replication can be used to build a highly available system, but it has to be 2-safe for that purpose.

The survey together with the experiments done for this thesis did not result in finding an open source database which would suit the needs of mobile telecommunications. Further work could be done in trying to develop the replication of MySQL 2-safe and analyzing deeper the performance degradation of InnoDB in disk-based use. The experiments in this thesis concentrated on a single server system, with a system built using 2-safe replication availability and multi-node scalability tests could be done as well.

References

AA02 Arnö, K. and Axmark, D., ACID Transactions in MySQL with InnoDB, Presentation in O'Reilly Open Source Convention, July 2002.

AS89 Agrawal, D. and Sengupta, S., Modular Synchronization in Multiversion Databases: Version Control and Concurrency Control. *Proceedings of the 1989 ACM SIGMOD international conference on Management of data.* ACM Press, 1989, pages 408–417.

BDB Berkeley DB, Sleepycat Inc. http://www.sleepycat.com, Aug. 7. 2002.

BG83 Bernstein, P. A. and Goodman, N., Multiversion Concurrency Control: Theory and Algorithms. *ACM Transactions on Database Systems (TODS)*, 8,4(1983), pages 465–483.

BGHJ92 Bhide, A., Goyal, A., Hsiao, H.-I. and Jhingran, A., An Efficient Scheme for Providing High Availability. *Proceedings of the 1992 ACM SIGMOD international conference on Management of data.* ACM Press, 1992, pages 236–245.

BHR80 Bayer, R., Heller, H. and Reiser, A., Parallelism and Recovery in Database Systems. *ACM Transactions on Database Systems (TODS)*, 5,2(1980), pages 139–156.

BK97 Breitbart, Y. and Korth, H. F., Replication and consistency: being lazy helps sometimes. *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems.* ACM Press, 1997, pages 173–184.

BKR+99 Breitbart, Y., Komondoor, R., Rastogi, R., Seshadri, S. and Silberschatz, A., Update propagation protocols for replicated databates. *Proceedings of the 1999 ACM SIGMOD international conference on Management of data.* ACM Press, 1999, pages 97–108.

Bra84 Bratbergsengen, K., Hashing Methods and Relational Algebra Operations. *Proceedings of the Tenth International Conference on Very Large Data Bases.* Morgan Kaufmann, 1984, pages 323–333.

BU77 Bayer, R. and Unterauer, K., Prefix B-Trees. *ACM Transactions on Database Systems*, 2,1(1977), pages 11–26.

CB02 Connolly, T. and Begg, C., *Database Systems.* Addison Wesley, third edition, 2002.

Com79 Comer, D., The Ubiquitous B-Tree. *Computing Surveys*, 11,2(1979), pages 121–137.

EMI Emic Networks, Emic Application Cluster for MySQL. http://www.emicnetworks.com, Apr. 5. 2003.

ETS ETSI, GSM 11.31: Home Location Register Specification. European Telecommunications Standards Institute (ETSI), Feb. 1992.

FB Firebird, Relational Database. http://firebird.sourceforge.net, Sep. 18. 2002.

GHOS96 Gray, J., Helland, P., O'Neil, P. and Shasha, D., The dangers of replication and a solution. *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*. ACM Press, 1996, pages 173–182.

GR93 Gray, J. and Reuter, A., *Transaction Processing: Concepts and Techniques*. Morgan Kaufman Publishers, Inc., 1993.

Gra93 Gray, J., editor, *The Benchmark Handbook for Database and Transaction Systems*. Morgan Kaufmann, second edition, 1993.

Gra94 Graefe, G., Sort-Merge-Join: An Idea Whose Time Has(h) Passed? *Proceedings of the Tenth International Conference on Data Engineering*. IEEE Computer Society, 1994, pages 406–417.

Her87 Herlihy, M., Concurrency versus availability: atomicity mechanisms for replicated data. *ACM Transactions on Computer Systems (TOCS)*, 5,3(1987), pages 249–274.

HR83 Haerder, T. and Reuter, A., Principles of Tansaction-Oriented Database Recovery. *ACM Computing Surveys (CSUR)*, 15,4(1983), pages 287–317.

IBP IBPhoenix Community. http://www.ibphoenix.com, Sep. 18. 2002.

IBR IBReplicator, The InterBase Replication Solution by Replicant Technologies. http://www.replication.co.za, Sep. 18.2002.

IDB InnoDB, Innobase Oy. http://www.innodb.com, Aug. 7. 2002.

Inn02 Innobase OY, http://www.innodb.com/ibman.html, *Reference Manual of InnoDB*, July 2002.

KA98 Kemme, B. and Alonso, G., A Suite of Database Replication Protocols based on Group Communication Primitives. *International Conference on Distributed Computing Systems*, 1998, pages 156–163, URL `citeseer.nj.nec.com/kemme98suite.html`.

KA00 Kemme, B. and Alonso, G., A new approach to developing and implementing eager database replication protocols. *ACM Transactions on Database Systems (TODS)*, 25,3(2000), pages 333–379.

Kim84 Kim, W., Highly available systems for database applications. *ACM Computing Surveys (CSUR)*, 16,1(1984), pages 71–98.

Lar88 Larson, P.-k., Dynamic Hash Tables. *Communications of the ACM*, 31,4(1988), pages 446–457.

Lit80 Litwin, W., Linear hashing: A new tool for file and table addressing. *Proceedings of the Sixth International Conference on Very Large Data Bases*. IEEE Computer Society, 1980, pages 212–223.

LLSG92 Ladin, R., Liskov, B., Shrira, L. and Ghemawat, S., Providing high availability using lazy replication. *ACM Transactions on Computer Systems (TOCS)*, 10,4(1992), pages 360–391.

Lom93 Lomet, D. B., Key Range Locking Strategies for Improved Concurrency. *19th International Conference on Very Large Data Bases*. Morgan Kaufmann, 1993, pages 655–664.

ME92 Mishra, P. and Eich, M. H., Join processing in relational databases. *ACM Computing Surveys (CSUR)*, 24,1(1992), pages 63–113.

MHL+92 Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H. and Schwarz, P., ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems (TODS)*, 17,1(1992), pages 94–162.

ML92 Mohan, C. and Levine, F., ARIES/IM: an Efficient and High Concurrency Index Management Method Using Write-Ahead Logging. *Proceedings of the 1992 ACM SIGMOD international conference on Management of data*. ACM Press, 1992, pages 371–380.

Moh90 Mohan, C., ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes. *Proceedings of the 16th International Conference on Very Large Data Base*. Morgan Kaufmann, 1990, pages 392–405.

Moh96 Mohan, C., Concurrency Control and Recovery Methods for B+-Tree Indexes: ARIES/KVL and ARIES/IM, *Performance of Concurrency Control Mechanisms in Centralized Database Systems* Prentice Hall, 1996, pages 248–306, 1996.

MPL92 Mohan, C., Pirahesh, H. and Lorie, R., Efficient and flexible methods for transient versioning of records to avoid locking by read-only transactions. *Proceedings of the 1992 ACM SIGMOD international conference on Management of data*. ACM Press, 1992, pages 124–133.

MyS MySQL AB. http://www.mysql.com, Aug. 7. 2002.

OBS99 Olson, M. A., Bostic, K. and Seltzer, M., Berkeley DB. *FREENIX Track: 1999 USENIX Annual Technical Conference*. Sleepycat Software, Inc., USENIX, June 1999, pages 183–192.

ODB Open Database Connectivity (ODBC). http://msdn.microsoft.com/library/default.asp?url=/library/en-us/odbc/htm/odch01pr.asp, Mar. 10. 2003.

PG PostgreSQL Inc. http://www.pgsql.com, Sep. 18. 2002.

Reu84 Reuter, A., Performance Analysis of Recovery Techniques. *ACM Transactions on Database Systems (TODS)*, 9,4(1984), pages 526–559.

Sal96 Salzberg, B., Access methods. *ACM Computing Surveys (CSUR)*, 28,1(1996), pages 117–120.

SAP SAPDB, Free Open Source Database. http://www.sapdb.org, Sep. 18. 2002.

Sha86 Shapiro, L. D., Join processing in database systems with large main memories. *ACM Transactions on Database Systems (TODS)*, 11,3(1986), pages 239–264.

SKS97 Silberschatz, A., Korth, H. F. and Sudarshan, S., *Database System Concepts*. The McGraw–Hill Companies, Inc., third edition, 1997.

Sle Sleepycat Inc., http://www.sleepycat.com/docs/reftoc.html, *Berkeley DB Reference Guide*, fourth edition.

SO92 Seltzer, M. and Olson, M., LIBTP: Portable, Modular Transactions for UNIX. *USENIX Conference Proceedings*. Univeristy of California, Berkeley, USENIX, Winter 1992, pages 9–26.

SO99 Seltzer, M. and Olson, M., Challenges in Embedded Database System Administration. *First Workshop on Embedded Systems*. Sleepycat Software, Inc., March 1999.

SQL SQLite, An Embeddable SQL Database Engine. http://www.hwaci.com/sw/sqlite/index.html, Oct. 4. 2002.

SSL+83 Stonebraker, M., Stettner, H., Lynn, N., Kalash, J. and Guttman, A., Document Processing in a Relational Database System. *ACM Transactions on Office Information Systems*, 1,2(1983), pages 143–158.

Sto86 Stonebraker, M., The Case for Shared Nothing. *IEEE Database Engineering Bulletin*, volume 9, 1986, pages 4–9.

SY91 Seltzer, M. and Yigit, O., A New Hashing Package for UNIX. *USENIX Conference Proceedings*. Univeristy of California, Berkeley; York University, USENIX, January 1991, pages 173–184.

Tho98 Thomasian, A., Concurrency control: methods, performance, and analysis. *ACM Computing Surveys (CSUR)*, 30,1(1998), pages 70–119.

Tik02 Tikkanen, M. J., Network Database Benchmark. *Internal Document*. NOKIA Networks, September 2002, pages 1–41.

TPC Transaction Processing Performance Council. http://www.tpc.org, Jan. 27. 2003.

Ver78 Verhofstad, J. S. M., Recovery Techniques for Database Systems. *ACM Computing Surveys (CSUR)*, 10,2(1978), pages 167–195.

WPS+00 Wiesmann, M., Pedone, F., Schiper, A., Kemme, B. and Alonso, G., Understanding Replication in Databases and Distributed Systems. *Proceedings of 20th International Conference on Distributed Computing Systems (ICDCS'2000)*. IEEE Computer Society Technical Commitee on Distributed Processing, 2000, pages 264–274, URL `http://lsewww.epfl.ch/Publications/ById/224.html`.

Appendix 1. HLR Database CREATE TABLE-definitions

```
CREATE TABLE subscriber (
        s_id INTEGER NOT NULL PRIMARY KEY,
        sub_nbr VARCHAR(15) NOT NULL UNIQUE,
        bit_1 TINYINT,
        bit_2 TINYINT,
        bit_3 TINYINT,
        bit_4 TINYINT,
        bit_5 TINYINT,
        bit_6 TINYINT,
        bit_7 TINYINT,
        bit_8 TINYINT,
        bit_9 TINYINT,
        bit_10 TINYINT,
        hex_1 TINYINT,
        hex_2 TINYINT,
        hex_3 TINYINT,
        hex_4 TINYINT,
        hex_5 TINYINT,
        hex_6 TINYINT,
        hex_7 TINYINT,
        hex_8 TINYINT,
        hex_9 TINYINT,
        hex_10 TINYINT,
        byte2_1 SMALLINT,
        byte2_2 SMALLINT,
        byte2_3 SMALLINT,
        byte2_4 SMALLINT,
        byte2_5 SMALLINT,
        byte2_6 SMALLINT,
        byte2_7 SMALLINT,
        byte2_8 SMALLINT,
        byte2_9 SMALLINT,
        byte2_10 SMALLINT,
        msc_location INTEGER,
        vlr_location INTEGER
) [TYPE=INNODB/BDB]

CREATE TABLE access_info (
        s_id INTEGER NOT NULL,
        ai_type TINYINT NOT NULL,
        data1 SMALLINT,
        data2 SMALLINT,
        data3 CHAR(3),
        data4 CHAR(5),
```

```
        PRIMARY KEY (s_id, ai_type),
        FOREIGN KEY (s_id) REFERENCES subscriber (s_id)
) [TYPE=INNODB/BDB]

CREATE TABLE special_facility (
        s_id INTEGER NOT NULL,
        sf_type TINYINT NOT NULL,
        is_active TINYINT NOT NULL,
        error_cntrl SMALLINT,
        data1 SMALLINT,
        data2 CHAR(5),
        PRIMARY KEY (s_id, sf_type),
        FOREIGN KEY (s_id) REFERENCES subscriber (s_id)
) [TYPE=INNODB/BDB]

CREATE TABLE call_forwarding (
        s_id INTEGER NOT NULL,
        sf_type TINYINT NOT NULL,
        start_time TINYINT NOT NULL,
        end_time TINYINT,
        number VARCHAR(15),
        PRIMARY KEY (s_id, sf_type, start_time),
        FOREIGN KEY (s_id, sf_type) REFERENCES special_facility(s_id, sf_type)
) [TYPE=INNODB/BDB]
```

Appendix 2. SQL Clauses of the Transactions

```
GET_BASIC_SUBSCRIBER_DATA
        SELECT  s_id,sub_nbr,
                bit_1,bit_2,bit_3,bit_4,bit_5,bit_6,bit_7,bit_8,bit_9,bit_10,
                hex_1,hex_2,hex_3,hex_4,hex_5,hex_6,hex_7,hex_8,hex_9,hex_10,
                byte2_1,byte2_2,byte2_3,byte2_4,byte2_5,
                byte2_6,byte2_7,byte2_8,byte2_9,byte2_10,
                msc_location,vlr_location
        FROM subscriber WHERE s_id = ?

GET_NEW_DESTINATION
        SELECT  cf.number
        FROM    special_facility AS sf,
                call_forwarding AS cf
        WHERE   (sf.s_id = ? AND sf.sf_type = ? AND sf.is_active = 1)
                AND (cf.s_id = sf.s_id AND cf.sf_type = sf.sf_type)
                AND (cf.start_time <= ? AND ? < cf.end_time)

GET_ACCESS_DATA
        SELECT  data1, data2, data3, data4
        FROM    access_info
        WHERE   s_id = ? AND ai_type = ?

UPDATE_SUBSCRIBER_DATA_1
        UPDATE subscriber SET bit_1 = ? WHERE s_id = ?
UPDATE_SUBSCRIBER_DATA_2
        UPDATE special_facility SET data1 = ? WHERE s_id = ? AND sf_type = ?

UPDATE_LOCATION
        UPDATE subscriber SET vlr_location = ? WHERE sub_nbr = ?

INSERT_CALL_FORWARDING_1
        SELECT s_id FROM subscriber WHERE sub_nbr = ?
INSERT_CALL_FORWARDING_2
         INSERT INTO call_forwarding VALUES (?,?,?,?,?)

DELETE_CALL_FORWARDING_1
        SELECT s_id FROM subscriber WHERE sub_nbr = ?
DELETE_CALL_FORWARDING_2
        DELETE  FROM call_forwarding
        WHERE   s_id = ?
                AND sf_type = ?
                AND start_time = ?
```