dsmDB: Clustering in-memory Database Management Systems

Master's Thesis submitted to the
Faculty of Informatics of the University of Lugano
in partial fulfillment of the requirements for the degree of
Master of Science in Informatics
Dependable Distributed Systems

presented by
Daniele Sciascia

under the supervision of
Prof. Fernando Pedone

August 2009

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Daniele Sciascia
Lugano, 24 August 2009

Abstract

Database management systems used in practice inherit their design from the early work of the database community. At the time, systems had limited memory and processing resources, and architectures were mainly centralized. Databases are often the “bottleneck” of performance-critical systems because of their heavy use of stable storage and mechanisms that allow concurrent transactions to be correctly executed.

In this thesis we investigate the dsmDB approach for clustering in-memory databases. The dsmDB is designed for distributing database computation and storage over a cluster of machines. Performance is enhanced by emphasizing in-memory computation and minimizing disk use. This is achieved by using an optimistic concurrency control mechanism on top of an in-memory storage layer that guarantees only weak consistency. By combining the two components we achieve both high performance and strong consistency.

The resulting architecture is also flexible enough to allow recovery of the state of crashed nodes from the state of the alive nodes, and incremental expansion by adding more nodes at runtime.

Acknowledgements

I thank my advisor Professor Fernando Pedone, and those who have worked on the dsmDB project, Marco Primi and Nicolas Schiper.

Contents

Figures

Tables

Chapter 1

Introduction

1.1 Problem statement

In performance-critical applications, databases are often the “bottleneck” of the system. For instance, consider popular web applications that are hit by millions of requests per day. Traditional database management systems heavily rely on disk storage and therefore, are not able to deal with such loads. A common solution to avoid this problem is the use of in-memory caching layers, which require additional software and machines that are responsible for caching query results in main memory. The context of web applications is just one example, similar solutions are employed in other contexts where databases need to scale.

In the spirit of [SMA+07], we investigate the need of new database architectures for the online transaction processing (OLTP) market. Databases commonly used in practice are based on the early work of the database community and derive their design decisions from it. Traditional databases are based on the requirements of thirty years ago, where databases needed to be stored on centralized architectures, main memory capacity and processor performance were extremely limited, and the concept of *scaling* a system was to replace the current machine with a “bigger” one. These requirements do not belong to the present. In recent years main-memory and processor technologies have evolved, and conceptually *scaling* means adding more machines to the system.

In this thesis we investigate the dsmDB approach [FP08] for clustering in-memory database management systems. The dsmDB approach assumes a modern clustered environment, where nodes have a significant amount of main memory and are connected to each other through fast network links. As opposed to traditional architectures for database systems, which are commonly used in practice, the dsmDB approach tries to take advantage of modern equipment in shared-nothing environments.

1.2 Building blocks

The dsmDB is based on the Distributed Shared Memory (DSM) paradigm [NL91]. The idea is to abstract shared memory on a networked architecture. Each node in a cluster shares its memory and a DSM mechanism conveniently allows for shared memory programming. If a process accesses a region of shared memory which is not stored locally, the DSM mechanism transparently retrieves it from the memory of a remote node. In principle, DSM should improve performance because fetching memory chunks remotely through fast network links is faster than storing and accessing data on a local disk. The shared memory programming paradigm is convenient, but it is difficult to design a DSM that provides both consistency and efficiency at the same time. Among the many consistency criteria that have been defined, *strict consistency* is perhaps the most intuitive but also the most inefficient to implement. In *strict consistency* a read operation always returns the last value written. While DSM systems that provide weaker levels of consistency perform better, they are not suitable for certain applications that depend on stronger forms of consistency.

The dsmDB approach combines an efficient DSM layer that provides weak consistency, and is augmented with a concurrency control mechanism. This mechanism has two important features: (1) it is optimistic; and (2) it works over versioned data items. Optimistic concurrency control allows concurrent transactions to be freely executed without synchronizing them. At commit time, transactions are submitted to a *certifier* that validates them and aborts conflicting. To increase the degree of multiprogramming, the certifier takes into account multiple version of data items. The *certifier* finally uses a broadcasting algorithm to send transaction batches back to nodes.

DsmDB transactions are implemented on top of these building blocks. Concurrent transactions are executed without synchronization over the data set stored in shared memory. The shared memory mechanism provides only a weak level of consistency, therefore at commit time a concurrency control mechanism ensures global consistent execution.

1.3 Design considerations

1.3.1 Minimizing disk use

For modern server machines it is common to have several gigabytes of main memory. For most OLTP workloads, the aggregated memory of multiple servers arranged in a shared-nothing environment are sufficient to store the whole data set in memory. Retrieving data from a remote node through fast networks is more efficient than retrieving data from a local disk.

1.3.2 Synchronization

Traditional database management systems include complex synchronization mechanisms and concurrent data structures. We completely avoid these complications since OLTP transactions are typically short and local execution is done instantly. Still, the distributed nature of the dsmDB allows for high degrees of multiprogramming over an optimistic concurrency control mechanism at the transactional level.

1.3.3 High availability

To achieve high availability it is still common to run *hot standby* machines. That is, the main machine does all the work and a hot standby machine sits idle until the main machine fails. We prefer a distributed approach, where all machines are used at the same time, collaborating and working together. In such an environment the failure of a machine should only cause temporary degraded performance.

1.3.4 Incremental recovery

In traditional database management systems recovery is based on some disk-based logging mechanism. For each running transaction, the system logs the operations executed on the database. If the system crashes, undo and redo logs allow to respectively undo unfinished transactions and persistently redo completed transactions. During the recovery procedure the database cannot perform any operations until its state is completely recovered. The dsmDB approach allows for incremental recovery: if data items are replicated at multiple nodes, a crashed node may recover its state by fetching its data items remotely from the other nodes during normal transaction execution.

1.3.5 Incremental expansion

The dsmDB approach allows incremental expansion. A running dsmDB instance may reconfigure itself allowing additional nodes to join the instance. It is important for such a system to handle the addition of nodes so that more memory is available if the data set increases, or to achieve better performance if the number of client requests increases. Reconfiguration in the dsmDB is done on-the-fly, without disrupting normal execution (i.e.,, no "fork-lift" approach to reconfiguration [SMA$^+$07])

1.4 Related Work

So far we introduced the design considerations behind the dsmDB approach, which we share with [SMA+07]. In this work, we demonstrate a practical use of Distributed Shared Memory [NL91] over weak consistency. Many practical alternatives to database management systems have been defined and implemented. We introduce some of them in the rest of this section.

The dsmDB approach resembles the Database State Machine [PGS03], with two main differences: (1) the Database State Machine maintains fully replicated data at every node; and (2) in the Database State Machine every node broadcasts transactions one by one and every node maintains its *certifier*. The dsmDB approach is optimized for in-memory processing and does not maintain full replication at each node. Moreover, the dsmDB *certifier* is a separate task that collects transactions coming from nodes and broadcasts transaction batches, thus minimizing the number of broadcast operations.

Dynamo [HJK+07] is a distributed key-value store developed and in use at Amazon. Although the dsmDB and Dynano share a similar query interface, made of *read* and *write* operations, their semantics differ. In the dsmDB, if two concurrent transactions conflict, one of them must be aborted, whereas Dynamo is based on *eventual consistency*, where transactions are never aborted but isolation is not guaranteed by the system. Conflict resolution is moved to the client side, meaning that clients need to handle conflicts by reconciling conflicting versions of data items. This mechanism ensures high availability at the cost of ease of use.

Sinfonia [AMS+07] offers a new paradigm for building scalable distributed systems. Typical full fledged transactions are replace by *minitransactions* over unstructured data. Minitransactions, are less expressive than SQL transactions, but can be quickly processed and enable an optimized two-phase commit protocol that does not block on coordinator crashes. Thus, Sinfonia allows to build distributed systems on top of a simplified query interface. However load balancing is not automatically handled by the system, because applications must decide where to place data while using minitransactions. As a consequence, Sinfonia cannot automatically handle hotspots in the workload, by placing related data in the same server.

The authors of [SMA+07] also describe a system called H-Store which shares common design considerations with the dsmDB approach. In H-Store transaction execution is optimized depending on transaction classes and schema characteristics. For instance, it is not clear how to distinguish *two-phase*, *strictly two-phase* and *sterile* transactions programmatically. As a consequence transaction classes need be declared a priori, which makes the system less flexible. Moreover, H-Store does not address the problem of reconfiguration.

1.5 Thesis outline

Chapter 2 presents the architecture and algorithms behind the dsmDB approach. We start with a description of the system model and the architecture and its components. Follow discussions on concurrency control, partitioning and finally recovery and reconfiguration procedures.

Chapter 3 discusses a prototype implementation of the dsmDB. Here, the discussion focuses on implementation decisions and details of practical interest.

Chapter 4 presents benchmarks we used to evaluate our prototype. The rest of the chapter presents performance results and evaluation.

Chapter 5 concludes the thesis and outlines possible future work.

Chapter 2

The dsmDB approach

2.1 System Model

We define dsmDB as a distributed key-value store composed of N nodes. Nodes are responsible for storing a given data set. A data set is composed of data items, and each data item is a pair (k, v), where k is a key associated to a value v. Both k and v are treated as arbitrary byte arrays, no structure is imposed on the data. Nodes are not required to use stable storage, the data set is stored over the aggregated main memory of nodes.

Clients operate on the data set through a simplified transactional query interface. A transaction in dsmDB is a sequence of *read* and *write* operations, possibly spanning over multiple data items. Transactions in the dsmDB are ACID, although durability relies on the assumption that a certain number of nodes do not fail.

There are two different ways for nodes to communicate with each other, point-to-point and atomic broadcast: the first can be used for sending messages directly from one node to the other through simple *send* and *receive* primitives (we assume nodes all "know" each other); the second can be used to broadcast a message from one node to a set of other nodes through *broadcast* and *deliver* primitives. These primitives are useful for spreading transaction updates and for ensuring consistent transaction execution on all nodes. Atomic broadcast guarantees agreement and total order:

Agreement If node n delivers a message m, then all non-faulty nodes eventually deliver m.

Total order If n and n' both deliver messages m and m', then n delivers m before m' if and only if n' delivers m before m'.

In the context of the described system model, we also investigate some additional and desired properties. One such property is *high availability*, the ability for a system to serve requests despite a failure. Another interesting property to consider is *incremental expansion*, the ability to add nodes while the system is running, so that it can handle a growing data set or a growing number of client requests without stopping the execution (i.e., no "forklift" upgrades).

2.2 System Architecture

The architecture of the dsmDB is composed of clients, multiple dsmDB nodes, and a consistency manager. Figure 2.1 shows an instance of the system with two dsmDB nodes, and clients. DsmDB nodes store the data set and process transactions issued by clients, while the Consistency Manager ensures consistent execution of concurrent transactions. In this section we will have a closer look at these components and explain what they do.

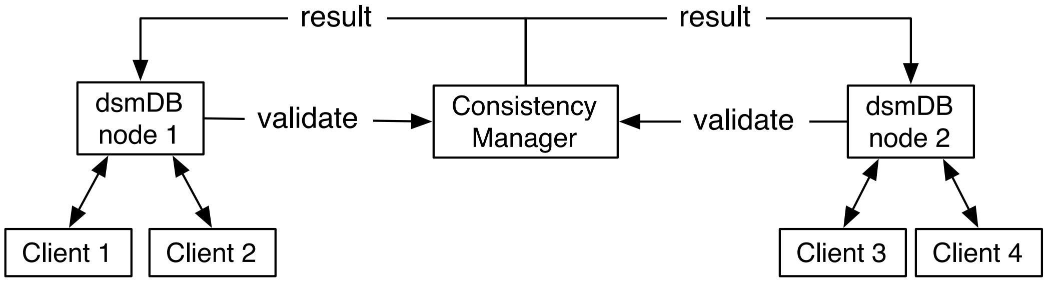

Figure 2.1: Architecture of the system.

2.2.1 Clients

From a client's point of view, the dsmDB is a data store that enables transactional access to data. Clients can store and retrieve data through a simplified query interface:

get(k): This primitive returns the value associated to the given key k.

put(k, v): This primitive stores the given key value pair (k, v).

commit(): This primitive commits the current transaction to dsmDB.

rollback(): This primitive aborts the current transaction.

The interface provided by the dsmDB is simplified and, as opposed to traditional DBMSs, values are treated as byte arrays and no structure or schema is imposed on the data; and it does not provide complex queries such as *select* constructs. A transaction in the dsmDB is simply a sequence of `get` and `put` operations followed by a `commit` operation. As in traditional DBMSs, a dsmDB transaction execution is guaranteed to be ACID:

Atomicity: Transactions are executed atomically, meaning that either all of the operations of a transaction are executed or none of them are.

Consistency: If the state of the database was consistent before the execution of a transaction, then the state remains consistent also after its execution.

Isolation: Transactions do not see intermediate results of other transactions. A transaction has the "impression" to be the only one executing.

Durability: Once a transaction is reported to be committed, its modifications will persist in the database.

2.2.2 DsmDB nodes

In the dsmDB architecture, every node shares its main memory and is responsible for storing a portion of the data set. Figure 2.2 shows that a single dsmDB node is composed of a Transaction Manager, a Broadcast Manager and an In-Memory Storage.

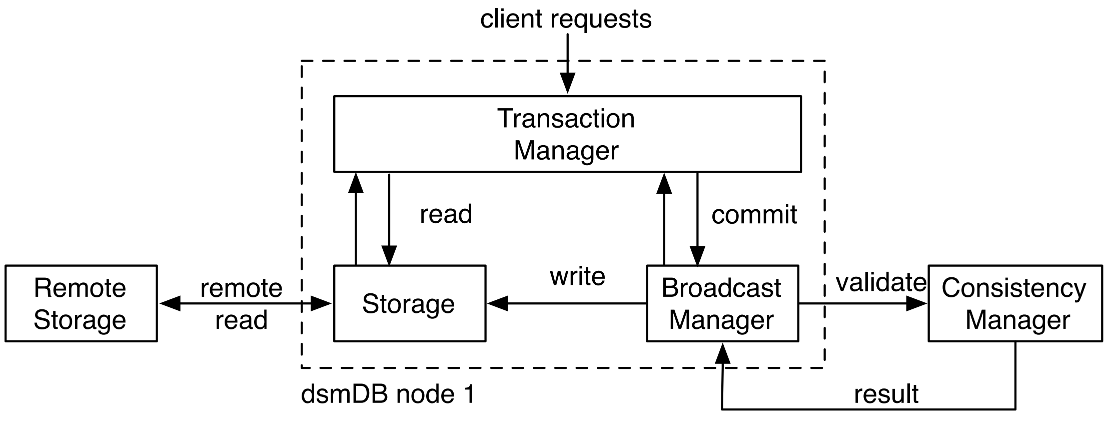

Figure 2.2: DsmDB node internals.

2.2.3 Transaction Manager

The Transaction Manager is mainly responsible for executing client requests. For each client, it keeps a *transaction record* that represents the client's currently executing transaction. Transaction records are intended to provide isolation and to keep track of:

Read set the set of keys $\{k_1, k_2, \ldots, k_n\}$ read by the transaction; and

Write set the set of pairs $\{(k, v)_1, (k, v)_2, \ldots, (k, v)_n\}$ written by the transaction.

On a `put` request, the value and its associated key are stored in the write set of the transaction record. On a `get` request, the key is stored in the read set of the transaction record. To complete the `get` request, the value associated to the given key is either fetched from the Storage, or from the transaction record if that key was previously written by the current transaction. On a `commit` request, the transaction record is given to the Broadcast Manager. Eventually, a reply from the Broadcast Manager will tell the Transaction Manager that the transaction was either committed or aborted. At that point the Transaction Manager will also send the reply to the client that executed the transaction in the first place.

2.2.4 Storage

The Storage component is based on the concept of Distributed Shared Memory [NL91]. Each dsmDB node is required to store a portion of the database in its Storage component. The Storage component provides simple *read* and *write* operations on single data items. These operations give the illusion that the Storage contains the whole data set, which is not necessarily true because the whole data set might not even fit into memory. If a read operation tries to access a key which is not present in the Storage, the read operation is turned into a *remote read* request sent to the Storage of the dsmDB node which is responsible for storing that key. Notice that this can be efficiently implemented because these operations must provide only a weak level of consistency, so fetching a remote item requires just a single round of point-to-point communication. We will discuss partitioning and key ownership in Section 2.4.

The Storage is updated whenever a transaction has been reported to be committed. In these cases, the write operation is required to store only those objects for which the Storage is responsible for. If space permits, the write operation heavily caches objects to avoid future *remote read* requests. As we will see in Section 2.3, the Storage must be able to handle multiple versions and is allowed to return stale data from its cache.

2.2.5 Consistency Manager

The Consistency Manager is a key component for ensuring global consistency among every dsmDB node. In Section 2.2.4 we described the Storage as a component that provides weak consistency. This means that concurrent transactions executing on different nodes are not synchronized. DsmDB provides an optimistic concurrency control mechanism, meaning that: transactions are free to execute on dsmDB nodes without worrying about consistency and concurrency, at commit time some concurrency control mechanism will take care of checking whether concurrent executions are consistent.

Recall from section Section 2.2.3 that when a client commits a transaction, the Transaction Manager passes the transaction record to the Broadcast Manager, which in turn sends it to the Consistency Manager. The Consistency Manager collects incoming transactions and *validates* them.

The validation algorithm is similar to the one employed in the Database State Machine approach [PGS03]. It ensures one-copy serializability, that is, the execution of concurrent transactions on different nodes should be equivalent to a serial execution on a single node. If two transactions read an object, and at least one of them has also modified the object, then these transactions conflict. When two concurrent transactions conflict, one of them must be aborted. We will further discuss and give examples of one-copy serializable schedules in Section 2.3.1.

Validated transactions are placed in a buffer and each of them is marked as either to abort or to commit. The validation buffer is periodically broadcast to all dsmDB nodes. More precisely, the buffer has a fixed size and it is broadcast when: (a) the buffer is full i.e., the number of transactions in the buffer is equal to its size; or (b) the buffer size (in bytes) has reached the maximum size allowed by the broadcast primitive; or (c) periodically when a timeout occurs.

Broadcast primitives are implemented by the Paxos algorithm, which we will explain in Section 3.2. Messages broadcast through Paxos are guaranteed to be delivered in the same order in the Broadcast Managers of all dsmDB nodes. When a Broadcast Manager receives such a message, it will go through each committed transaction and write to Storage its updates (recall that the Storage module might discard objects for which it is not responsible). The Broadcast Manager also notifies the Transaction Manager the final decision of a given transaction so that in turn the Transaction Manager can reply to the client with the transaction's outcome, either committed or aborted.

2.3 Concurrency control

So far we discussed the dsmDB by describing its architecture and the components it is made of. It remains to establish how exactly transactions are consistently executed. In this section we introduce the notion of one-copy serializability and how it is implemented in dsmDB.

2.3.1 One-copy serializability

DsmDB uses an optimistic multiversion concurrency control mechanism. Optimistic, meaning that different processes are allowed to execute transactions inconsistently, and in a second step a certifier makes sure that conflicting transactions are aborted. Multiversion because each write on a data item produces a new version. Therefore, the storage has to keep multiple versions of a data item and transactions are allowed to ask for specific versions. Keeping multiple versions of a data item has the benefit of increasing the degree of multiprogramming.

The consistency criteria adopted in the dsmDB approach is one-copy serializability [BHG86]. A sequence of transactions executed in a distributed setting is one-copy serializable if it is equivalent to a serial schedule executed sequentially on a single machine. Denote $r_i(x_j)$ as "transaction i reads version j of value x" and $w_i(x_j)$ as "transaction i writes version j of value x", and consider Schedule 1 depicted in Figure 2.3.

Schedule 1: $w_0(x_0)$, $r_1(x_0)$, $w_1(x_1)$, $r_2(x_0)$

Figure 2.3: Schedule 1, a one-copy serializable schedule.

Even if T_2 reads version 0 of value x after $T1$ wrote version 1 of value x, Schedule 1 is one-copy serializable: we can create an equivalent serial schedule by rearranging the transactions so that each read returns the last value written. In fact, Schedule 1 is equivalent to Schedule 2 (Figure 2.4).

Schedule 2: $w_0(x_0)$, $r_2(x_0)$, $r_1(x_0)$, $w_1(x_1)$

Figure 2.4: Serial Schedule 2, equivalent to Schedule 1.

Consider Schedule 3 in Figure 2.5 as an example of a schedule which is not one-copy serializable. Transaction T_2 reads value x written by transaction T_0 and value y written by transaction T_1. Schedule 3 is not one-copy serializable because there is no way to rearrange those transactions in such a way that the execution is consistent.

Schedule 3: $r_0(x_0)$, $w_0(x_1)$, $w_0(y_1)$, $r_1(x_1)$, $w_1(x_2)$, $w_1(y_2)$, $r_2(x_1)$, $r_2(y_2)$

Figure 2.5: Schedule 3, a non one-copy serializable schedule.

2.3.2 Transaction execution

To allow only one-copy serializable schedules to be executed, concurrency control in the dsmDB must be handled by both dsmDB nodes and Consistency manager.

dsmDB node

Algorithm 1 sketches the functionality of dsmDB nodes. A dsmDB transaction is an object that keeps track of its read set $T.rs$, its write set $T.ws$ and its update set *T.us*. Every dsmDB node keeps track of its current *snapshot time*, an integer ST which is initially set to 1. As we will see later, dsmDB groups transactions according to their *snapshot time* and tries to commit them together. Every transaction also keeps a field *T.start*, which is used to mark the *snapshot time* in which the transaction started. Whenever a transaction starts, its read set, write set and update set are initialized to the empty set, while *T.start* is initially set to the special symbol $\perp$.

On a `get` operation, we check if *T.rs* is still empty, meaning that no `gets` have been executed yet. If so, *T.start* gets the value of the current *snapshot time ST*. Parameter k is the key to be read, but before reading it, it must be added to the read set. Finally, we read the value corresponding to the given key. We distinguish two cases: (1) key k was previously written by T, in which case we return the last version of the value written by T; (2) the key was not written by T, in which case the value is retrieved from the storage. Notice that *storageGet()*, takes as parameter also the *snapshot time* in which T started. That's because the storage handles multiple versions and (k, v) pairs are marked with the *snaphot time* in which they were written. *StorageGet(k, T.start)* will return the freshest version v of k such that $v \leq T.start$. A transaction that started at *snapshot time* ST, will never read values written at $ST' > ST$.

On a `put` operation, a dsmDB node keeps track of the (k, v) pair to be put. This means that we simply need to update read set and the update set of the transaction. Key k is added to *T.ws* and the key value pair (k, v) is added to *T.up*.

Algorithm 1: dsmDB node

```
Initialization
    ST ← 1

On start(T) do
    T.start ← ⊥
    T.rs ← ∅, T.ws ← ∅, T.us ← ∅

On get(T, k) do
    if $T.rs = \emptyset$ then
        T.start ← ST
    T.rs ← T.rs ∪ {k}
    if $k \in T.ws$ then
        v ← last version of k written by T
    else
        v ← storageGet(k, T.start)
    return v

On put(T, k, v) do
    T.ws ← T.ws ∪ {(k)}
    T.us ← T.us ∪ {(k, v)}

On commit(T) do
    if $T.ws = \emptyset$ then
        Report T as committed
    else
        Send T to Consistency Manager

On deliver(msg) do
    ST ← msg.ST
    foreach $T \in msg.transactions$ do
        if $T.commit = true$ then
            foreach $(k, v) \in T.us$ do
                storagePut(k, v, ST)
            Report T as committed
        else
            Report T as aborted
```

At commit time, we distinguish two cases: if T, the transaction to be committed, is read only (write set is empty), a dsmDB node can immediately report T as committed. Otherwise, if the transaction is not read only, T is sent to the Consistency Manager for validation. Notice that a `get` operation always reads data which is consistent with respect to its starting *snapshot time*, therefore read only transactions are always consistent and do not have to be validated.

Messages coming from the Consistency Manager are delivered using the broadcast primitive. Such a message contains: (1) The updated value of ST; (2) the set of transactions that the Consistency Manager received in the previous *snapshot*. When a message is delivered, nodes first update their current ST, and extract the set of transactions from the message. For each transaction, we check whether it is marked as committed or aborted. For each committed transaction T we update the storage with the key value pairs found in *T.us* and report T as committed. Notice that each node receives these messages in total order, meaning that every node delivers messages in the same order. Therefore each storage will be updated in the same order too. Finally, for each aborted transaction, we simply report the outcome to the client.

Consistency Manager

Algorithm 2 is a sketch of the code of the Consistency Manager. The Consistency Manager keeps track of the current *snapshot time ST*, the set of transactions aborted in the current ST in *abortBuffer*, the set of transactions committed in the current ST in *commitBuffer*, and the set of keys written at *snapshot time* $ST - 1$ in *previousWS*.

Whenever the Consistency Manager receives a transaction T, it checks *T.start*, the *snapshot time* in which the transaction started. Transactions that started at the current ST (for which $T.start = ST$ holds) are immediately validated. Older transactions need to be handled more carefully, as we will later discuss. Function *validate(T)* is responsible for inserting transactions in the commit buffer. A transaction T is inserted in the *commitBuffer* at position i if the following two statements hold:

1. For all transactions $T' \in \text{commitBuffer}[1, i-1]$: $T'.ws \cap T.rs = \emptyset$.

2. For all transactions $T' \in \text{commitBuffer}[i, \text{bufferSize}]$: $T'.rs \cap T.ws = \emptyset$.

Intuitively, $validate()$ tries to find a serial schedule in which there is no transaction that reads a value previously written by another transaction. Whenever $validate(T)$ is able to find a valid position i for T in the *commitBuffer*, T

Algorithm 2: Consistency Manager

```
Initialization
    ST ← 1
    abortBuffer ← null
    commitBuffer ← null
    previousWS ← null

function validate(T)
    if $\exists\, i \in [1, bufferSize]:$
            $\forall\, j \in [1, i)\ commitBuffer[j].ws \cap T.rs = \emptyset \wedge$
            $\forall\, j \in [i, bufferSize]\ commitBuffer[j].rs \cap T.ws = \emptyset$
    then
        T.commit = true
        insertAt(commitBuffer, T, i)
    else
        T.commit = false
        abortBuffer ← abortBuffer ∪ {T}

function newSnapshot()
    ST ← ST + 1
    msg.ST ← ST
    msg.transactions ← (commitBuffer ∪ abortBuffer)
    broadcast(msg)
    previousWS ← null
    foreach $T \in commitBuffer$ do
        previousWS ← previousWS ∪ T.ws
    abortBuffer ← null
    commitBuffer ← null

On receive(T) do
    validateOld ← false
    if $T.start = ST - 1 \wedge T.rs \cap previousWS = \emptyset$ then
        validateOld ← true
    if $T.start = \perp \vee T.start = ST \vee validateOld = true$ then
        validate(T)
        if isFull(commitBuffer) then
            newSnapshot()
    else
        T.commit = false
        abortBuffer ← abortBuffer ∪ {T}
```

is marked as committed and inserted into the *commitBuffer* at position i. If no such an i can be found, T is marked as aborted and appended to the *abortBuffer*.

The *commitBuffer* has a maximum size *bufferSize*, and when the Consistency Manager validates *bufferSize* transactions it will create a new snapshot. Function *newSnapshot()* does the following: it increments the current *snapshot time* ST and broadcasts a message containing the updated ST and the set of aborted and committed transactions to all nodes; it updates the set previousWS with all the keys that were written by the transactions in the *commitBuffer*; and finally it clears the *abortBuffer* and the *commitBuffer* so that the Consistency Manager is ready to process the next *snapshot time*.

So far we described how a transaction T is validated in the case *T.start* $= ST$. The Consistency Manager also handles transactions that started at *snapshot time* $ST - 1$, but arrived at the Consistency Manager at ST. The first if statement of *On receive(T)* handles exactly that case. Even if the buffer of *snapshot time* $ST - 1$ was already broadcast, there are chances that T, the transaction to be validated, has not read any key previously written by the transactions at $ST - 1$. That's why we keep *previousWS*, which is the set of keys that were written at $ST - 1$. If *T.rs* does not intersect $previousWS$, then it can be validated as if it was a transaction started at *snapshot time* ST. For the sake of simplicity, Algorithm 2 does not handle transactions that started at $ST - 2$, but it can be easily generalized to handle transactions arrived at $ST-$ *maxPreviousST*. To do so, it is not sufficient to keep track of just one *previousWS*; it needs to keep *previousWS*s of *snapshot time* $ST - 1$, $ST - 2$, ... , $ST -$ *maxPreviousST*.

2.3.3 Storage layer consistency

Algorithm 1 of Section 2.3.2 uses two important functions, *storageGet()* and *storagePut()*, that need to be explained more carefully.

Each data item in the Storage is tagged with a version number. In fact, *storagePut* takes the version number as argument when creating a new data item. A data item that was written by a transaction that started at *snapshot time ST* is tagged with version number *ST*. Similarly, when a transaction reads a data item using *storageGet()*, it will return a data item which is consistent with the *snapshot item* in which the transaction started.

In Section 2.2.4, we explained that the storage layer transparently handles remote lookups. It gives the impression that the whole data set is stored locally, whereas the data set might not even fit in a single machine. Function *storageGet()* implements exactly this behavior: if the data item to lookup is not stored locally, it will be fetched remotely from the node who is responsible for storing that data item. To improve the overall performance of the system, dsmDB nodes can cache items for which they are not responsible. There a few issues that arise from caching and remote lookups. We discuss those in the rest of this section.

Garbage collection

Each invocation of *storagePut()* creates a new version of the data item to be written in the storage. This means that each node accumulates old versions of data items and there must be a mechanism that collects old versions to make room for newer ones.

In Section 2.3.2 we said that the Consistency Manager is willing to accept transactions that started at $ST - maxPreviousST$, older transaction are aborted. If the newest version of some data item x was tagged with version v, then all items x tagged with a version smaller that $v - maxPreviousST$ are useless, because either:

- The item will never be accessed again. This is especially true if during the execution nodes are all roughly in the same ST (no node starts transactions which are too old), and there are no long running transactions.
- The item will be accessed, but the transaction reading or writing it will be aborted at the Consistency Manager.

A garbage collection mechanism can safely remove all data items x such that $x.version < (ST - maxPreviousST)$. If some transaction T tries to access such a data item (locally or remotely), the node executing T can immediately abort it. Notice that old versions of a data item must be eliminated in order, from the oldest to the newest, without leaving "holes" in between. The reason for this restriction is explained in the next section.

Cache maintenance

Cached items should be treated as local items. When a node delivers updates, it can decide to cache items for which it is not responsible. These items are subject to versioning as local items, but they are marked as cached. The storage should implement some policy to evict cached items to make sure that there is always space for local items. A possible policy to adopt is LRU (*least recently used*): when eliminating some cached items, the storage chooses items that were the most least recently used. However, our Concurrency Control mechanism imposes a restriction: there should never be "holes" between versions of an item, otherwise a *storageGet()* would not always return the freshest version available, and transactions might be inconsistent. Therefore, for each cached data item d the following should be done:

- When an update is delivered for d, every node that caches d should either apply the update or otherwise remove d with all of its versions.
- When evicting d with version v, all versions $v' < v$ of d should also be removed.

Remote lookups

Remote get operations should be consistent, as if they were executed locally. The problem arises when nodes interleave the delivery of updates and remote gets. As an example, consider two nodes n_0 and n_1. Initially they are both at *snapshot time* $ST = 0$. Assume n_0 is responsible for item $x_0 = 0$ and n_1 for item $y_0 = 0$. Suppose the following transaction $T_1 : r(x_0), w(x_1), r(y_0), w(y_1)$ is executed at n_0. T_1 is delivered at n_0, but not yet at n_1. So, node n_0 is at $ST = 2$ and proceeds with $T_2 : r(x_1), w(x_2), r(y_1), w(y_2)$. If y is not cached at n_0, clearly y_1 is going to be fetched from n_1. If n_1 has not yet delivered T_1, then $r(y_1)$ will return version 0 of item y. From n_0's point of view, $r(y_1)$ did not return the freshest version of y, thus producing an inconsistent execution. To fix this, a remote get request includes the ST of the sender and the receiver must make sure they are at the same ST before replying.

On the other hand, if the receiver of a remote get request is at a later ST, the garbage collection mechanism might have removed the older requested value, as we just discussed. In such cases the receiver node replies with a message telling that the version requested is too old and the transaction that issued the remote get request must be aborted.

2.3.4 Session consistency

Notice that the concurrency control mechanism described in Algorithm 1 and Algorithm 2 may lead to the following situation: Client C executes transaction $T_1 : w(x = 1)$ on node 1, where x was initially set to 0. T_1 successfully commits and C immediately proceeds with transaction $T_2 : r(x)$ on node 2, which returns the previous value 0. Notice that if C submits T_1 and T_2 to the same node, then this kind of behavior is not possible because: (1) clients can submit only one transaction at a time (i.e., when committing, the client blocks until it receives the transaction outcome); and (2) it must be the case that $T_2.start > T_1.start$, therefore the storage cannot return the older version of x with value 0. However, our running example produces a valid one-copy serializable schedule if T_1 and T_2 are executed on two different nodes. If clients can submit transactions on different nodes, there is no guarantee that nodes deliver updates at the same "speed". If T_2 is submitted to a node which is at a previous snapshot with respect to the node where T_1 was executed, then T_2 might read older values. There are at least ways to ensure session consistency in the dsmDB:

1. Restrict clients so that they access only one node during the same session.
2. Clients keep track of the snapshot time of their last successfully committed transaction, so that they can refute transaction executions at earlier STs.

2.4 Partitioning and Replication

A solution to the partitioning problem determines which nodes are responsible for storing a certain data item. Desirable properties for data distribution are for example load balancing or hot spot avoidance. Meaning that each node in the system should be responsible for storing a balanced portion of the data set, and in such a way that the most accessed items are not placed in the same node.

High availability, and in our particular case also durability, can be achieved through data replication. With data replication we want to provide a given level of K-safety, meaning that each (k, v) pair is replicated over K different nodes. This way, even if K-1 nodes fail, there is at least 1 replica left which is responsible for a particular data item.

In this section we briefly describe and compare some alternative solutions to distribute and partition a data set across different nodes in terms of implementation issues, performance and memory overhead requirements.

2.4.1 Hash based

A naive solution is to distribute data items deterministically according to a hash function h. For example, node n is responsible for storing key k if $h(k) \bmod N = n$. Replication can be achieved by storing replicas at $K - 1$ successive nodes. In our running example, if $K = 3$ and node n is responsible for storing key k, replicas can be stored at nodes $(n + 1) \bmod N$ and $(n + 2) \bmod N$.

This simple scheme has two advantages: it does not require memory to store routing information, and handling remote lookups is simple and efficient. Since every node agrees on the same hash function h, on a `get` request, if the given key k is not locally stored, the data item is simply fetched from node $n = h(k) \bmod N$ (only one communication round). In fact, in terms of storage, a hash based scheme is optimal.

The problem of this scheme is that it does not allow to explicitly control the placement of data. As one of the consequences, it leads to poor performance when handling system reconfigurations. Let N be the initial number of nodes, as soon as a new node wants to join the system N is increased by 1 and the mapping $h(k) \bmod N = n$ does not hold anymore (for almost every n, with high probability). This means that every key in the data set must be rehashed and redistributed across $N+1$ nodes according to the outcome of the hash function under modulo $N+1$.

2.4.2 Fully replicated key set

In this scheme, each data item is stored as a triple (k, v, r), where r is the set of node ids responsible for storing key k. We define the key set as the set of all (k, r) pairs contained in the data set. If the key set is fully replicated, every node knows the complete key-to-node mapping.

This scheme is more complicated but allows greater flexibility. Data can be distributed according to some predefined policy. The simplest one would be: key k is stored at node $h(k) \bmod N$ and its successors. More complex policies might try to uniform the balance between nodes, place data items frequently accessed together at the same node, or try to eliminate hot spots. In fact, in this case the mapping is not fixed, but can change over time.

As opposed to the hash-based solution, in this scheme the number of data items that must be transferred in case of reconfiguration is minimized, but in general the memory overhead to keep the key set fully replicated is prohibitive, as we will later demonstrate.

2.4.3 Compressed mapping

The idea here is that, instead of maintaining a fully replicated key set, every node maintains compressed information that represents the key set for every other node. For example, a bloom filter is a compressing data structure that allows to efficiently represent sets and supports membership queries [BM02].

In the context of dsmDB, the scheme would work as follows: a node n is responsible for storing key k if b_n (n's bloom filter) contains key k. Every time a new data item (k, v) is delivered, nodes check if there exits some b_i such that it contains k. If there exists such a b_i, then node i is responsible for storing key k (notice that because of *false positives*, it might be that more than one bloom filter contains key k); otherwise nodes must choose the node that will take responsibility for k and update the corresponding bloom filter. This decision must be taken deterministically by every node.

To extend this scheme to support replication, it suffices to change the above updating rule so that for every data item nodes make sure that there are at least K bloom filters that contain key k. Notice that this could be done considering *false positives*.

Remote lookups are simple and efficient: one round of communication. To find a node responsible for key k, it suffices to find a b_i such that it contains k. If such a b_i exists, i is the node storing key k. Otherwise k does not exist in the system. When reconfiguring the system, only a small portion of the key set must be transferred to the arriving node, but also updated bloom filters must be exchanged.

2.4.4 Distributed mapping

In this scheme, each node is responsible for a portion of the data set and a portion of the *mapping*. The mapping can be thought of as a table that relates keys to node ids. For each data item in the data set, the mapping contains an entry that consists of a pair (k, n), meaning that key k is stored at node n. The mapping itself is partitioned over the available nodes.

Node n is the *map owner* of key k, if $h(k) = n$, for some hash function h in $0 \ldots N-1$. The map owner of key k is responsible for storing the map entry for key k. Whereas node n' is the *key owner* of key k, if there exists some node in which $map(k) = n'$, where $map(k)$ is the map entry corresponding to k. The *key owner* of key k is responsible for storing the data item with key k.

It is straightforward to extend the above described mapping to support also replication. Multiple map owners can be decided in the following way: if node $n = h(k)$ is the map owner, then $n+1 \bmod N$, $n+2 \bmod N, \ldots, n+(K-1) \bmod N$ are map owners too. Multiple key owners are achieved by extending the map entries to map a key k to K different nodes ids.

Remote lookups can be done in at most three communication delays: the first round must reach a map owner, and the map owner forwards the request to a key owner, which in turn replies directly to the first node. Notice that this can be optimized: once the remote request reaches the map owner, instead of immediately forwarding the request to a key owner, the map owner can lookup its cache. If it cached the data item, it can immediately reply and the lookup takes only one round (i.e., two communication delays).

2.4.5 Memory overhead comparison

In the ideal case, partitioning and replication would be implemented efficiently and with no memory overhead for storing the mapping from keys to node ids. Let nk be the number of keys, ks the key size, vs the value size and rf the replication factor. The total amount of storage required is $nk \cdot rf \cdot (ks + vs)$. If n is the number of nodes available in the system and m the memory available at each node, then $n \cdot m$ is the total available memory in the system. Maximizing memory usage, we get the following equation:

$$n \cdot m = nk \cdot rf \cdot (ks + vs) \tag{2.1}$$

Following the same approach, we can compute the total amount of storage require by each of the schemes we discussed, and compare them against the optimal case.

Fully replicated key set

The mapping for this scheme additionally requires to store a table in which each row is a pair (key, nodes ids). The table is fully replicated and for each key there are rf node ids. Assuming that each node id is 4 bytes we have that the table requires $n \cdot nk \cdot (ks + 4 \cdot rf)$. Thus, we have:

$$n \cdot m = (nk \cdot rf \cdot (ks + vs)) + (n \cdot nk \cdot (ks + 4 \cdot rf)) \tag{2.2}$$

Bloom filter based

To compute the space overhead in the case of bloom filters, we need to know how much space each bloom filter requires. The false positive rate of a bloom filter is computed as follows [BM02]:

$$p \approx (1 - e^{\frac{-kn}{m}})^k$$

Where p is the false positive rate, k is the number of hash functions, n is the expected number of entries and m is the size of the bloom filter in bits. The optimal number of hash function is:

$$k \approx ln(2)\frac{m}{n} \approx 0.7\frac{m}{n}$$

As an example, assume we want to insert 100 keys in a bloom filter of 1000 bits. Then we have the optimal number of hash functions:

$$k = 0.7\frac{1000}{100} = 7$$

Which gives the probability of a false positive:

$$p \approx (1 - e^{-\frac{7 \cdot 100}{1000}})^7 \approx 0.008$$

In our example, this means we need only 10 bits (i.e., 1000 divided by 100) for every key, as opposed to the whole key, to maintain a bloom filter with a false positive rate of less than 1% (notice that bloom filters grow linearly in the number of elements inserted).

In the bloom filter based scheme, each node additionally stores a bloom filter for every other node. If we assume that the data set is balanced, each node should hold around $\frac{n}{nk}$ keys. Moreover, each node holds n bloom filters, each containing $\frac{nk}{n} \cdot rf$ keys, for a total of $n \cdot \frac{nk}{n} \cdot rf = nk \cdot rf$. It remains to consider that these bloom filters are replicated at each node, and that we need 10 bits (= 1.25 bytes) per key. Thus we have:

$$n \cdot m = (nk \cdot rf \cdot (ks + vs)) + (n \cdot nk \cdot rf \cdot 1.25) \tag{2.3}$$

Distributed mapping

In the case of distributed mapping, each node needs to additionally store a portion of a table in which each row is a pair (key, node ids). The table is not fully replicated, thus we have:

$$n \cdot m = (nk \cdot rf \cdot (ks + vs)) + (nk \cdot (ks + 4 \cdot rf) \tag{2.4}$$

Comparison

Figure 2.6 summarizes our comparison of the above schemes. It shows the ratio between each scheme and the optimal scheme. We compute different series for different value sizes. For instance, consider fully replicated key set when the number of nodes is 3 and value size is 16 bytes. The point shown is the ratio of key-value pairs that can be stored using an optimal scheme over the number of key-value pairs stored in the case of fully replicated key set. In this particular example the ratio is about 0.5, meaning that only half of the key-value pairs can fit in the database, because the rest of the space is needed to store the key set which is fully replicated on every node.

The memory available at each node m is fixed at 1GB, the key size ks is 8 bytes, and the replication factor rf is 3. These graphs essentially show that both the fully replicated key set and the bloom- filter-based techniques do not scale as the number of nodes increases. As the number of keys increases, the mapping or the bloom filters to store increase at every node, making it linear in the number of nodes. This is especially true in the extreme case where the value size (vs) is just 16 bytes. In these cases the number of entries to store in the fully replicated key set is prohibitive, and similarly the size required by the bloom filters becomes excessive. The bloom filter based technique improves over the fully replicated key set, because it stores a compressed representation of the mapping. The distributed mapping scheme improves the memory overhead

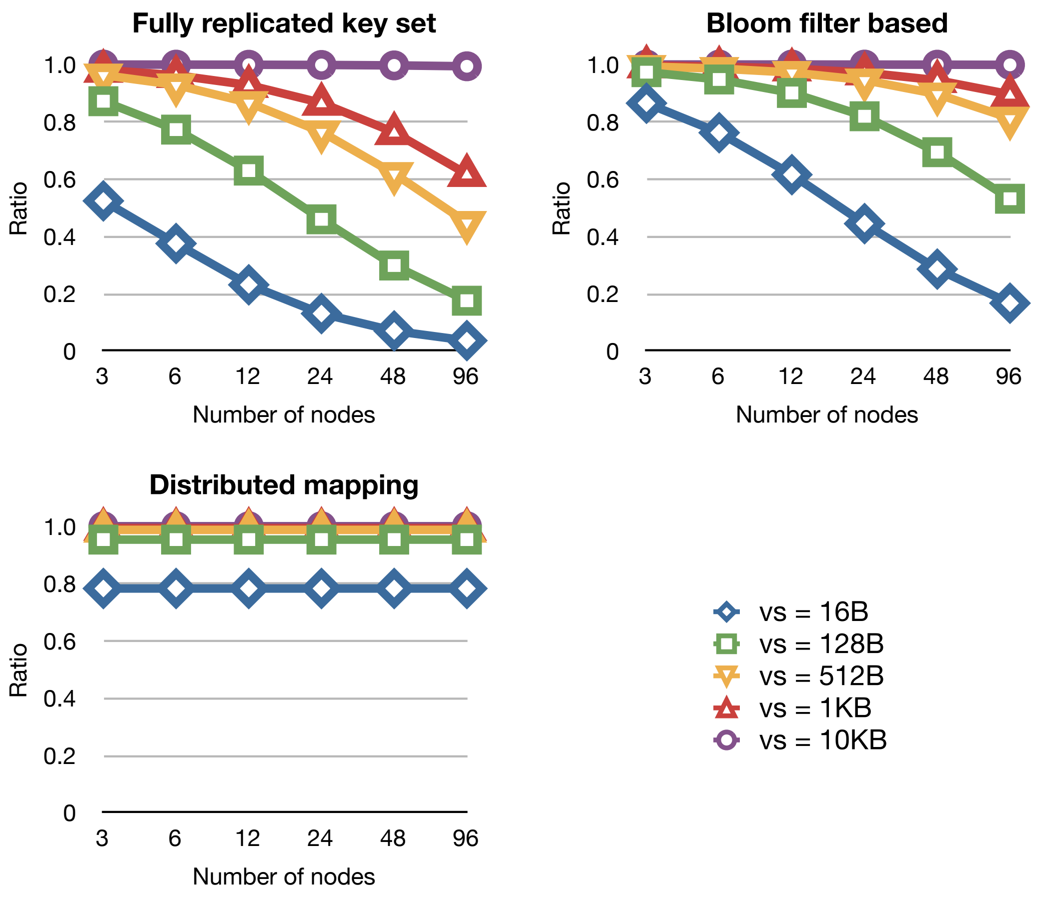

Figure 2.6: Memory overhead comparison.

because the memory required to store the mapping is constant in the number of nodes. DsmDB's partitioning and replication relies on the distributed mapping scheme because:

- it allows to freely control the placement of data, which is not possible when data items are statically placed as in the hash-based scheme. As a consequence of dynamic data placement we have a system where data items can possibly move from one node to another.
- it is the only scheme, among the ones we considered, that offers constant memory overhead in the number of nodes.
- it requires a bounded number of communication delays during a lookup (at most three). Schemes that require fewer communication delays, either place data statically, or impose prohibitive memory overhead.

2.5 Recovery

Traditional database management systems typically use some sort of log-based recovery (e.g., write-ahead logging). During transaction execution, the database logs modifications in a sequence of log records on stable storage. After a system crash, the recovery procedure ensures that the database recovers in a consistent state. The state of the database is first recovered from stable storage, and then the recovery procedure uses the log to undo unfinished transactions and to redo completed transactions. The database is ready to execute new transactions only after recovery. Log maintenance and recovery are expensive operations because they require constant use of stable storage.

The dsmDB approach allows for a simpler and lightweight recovery mechanism. Given that nodes do not use stable storage to store their portion of the data set, the state of the database is incrementally recovered from the other nodes. The state of a dsmDB node consists of its mapping table, which we discussed in Section 2.4.4, and its portion of the data set. A crashed node restarts with an empty storage and an empty map and can immediately start processing transactions coming from clients and delivering transaction updates. Data items read by these transactions are fetched remotely. Notice that to know the owner of a map, only the number of nodes in the system must be known. Initially, without the mapping from keys to node ids, a `get` operation might require up to four communication delays instead of three.

A crashed dsmDB node needs to retrieve both its map and its portion of the data set. To do so, it broadcasts a message that tells all the other nodes that it is recovering. The message contains the node id of the recovering node. We assume that the node that replaces the crashed node restarts with the same id, so that the mapping does not change. Nodes that receive a recovery message from node i scan their mapping and prepare two lists:

key owner list: a list containing all keys k such that $i \in map(k)$.

map owner list: a list containing all map entries for which i is responsible. Node i is responsible for $map(k)$ if $i \in \{h(k),\ h(k)+1 \bmod N,\ \dots,\ h(k)+(rf-1) \bmod N\}$, for some predefined hash function h.

These lists are eventually sent to recovering node i, which collects them and starts processing them in a separate task. It uses the map owner list to update its own map, and for each key k in the key owner list it fetches k remotely. Before fetching k in the key owner list, it checks if k is cached in the local storage; if so, k is marked as local and there is no need to fetch it remotely. Node i is fully recovered as soon as all keys in its key owner list are stored locally.

2.6 Reconfiguration

DsmDB can reconfigure itself, meaning that the number of nodes can increase or decrease, without stopping and restarting the system. A node joining a running instance of dsmDB starts with an empty storage and an empty map. In the following we discuss how to update the maps after the join of a new node.

Recall from Section 2.4.4 that in the case of distributed mapping, a node n is a map owner of key k if $n \in \{h(k),\ h(k)+1 \bmod N,\ \dots,\ h(k)+(rf-1) \bmod N\}$ for some predefined hash function h. When a new node joins a running dsmDB instance, N is increased by 1, and the map ownership changes according to the outcome of hash function h under modulo $N+1$.

If we assume that only one node joins a running instance of dsmDB at a time, then we can reconfigure the system without disrupting transaction execution. To do so, a node that is joining chooses a new id n, and broadcasts a join message to every other node with its new id. At this point, map entries can be exchanged lazily during transaction execution. The idea is that each node keeps all of its map entries until they are sure that the new owners received it.

A node that delivers a join message, first deletes all data items from its cache and exchanges map entries during transaction execution. Assume that some transaction T is executing at node i. If T reads key k, and i is not a key owner of k, the data item will not be found in the storage. Meaning that key k must be fetched remotely. The first step in the transaction execution is to create a remote get request and send it to one of the map owners. Since the number of nodes N is increased by 1, the map ownership changes, and the remote request might hit a map owner that doesn't have the map entry for k yet. If so, the new map owner will fetch the map entry for key k remotely from the previous map owner. As soon as the new map owner receives the map entry, it will broadcast a message that contains the map entry. To reduce the number of broadcast calls, nodes can batch multiple map entries in a single message. We distinguish two cases when a node delivers such message: new map owners for key k update their map by adding the map entry; old map owners for key k update their map by deleting the map entry. The reconfiguration process terminates when all nodes have updated their map, meaning that there are no map entries at node n for which n is not responsible. Notice that if there is some key k that is never accessed, the reconfiguration process will not terminate because the map entry for k remains at the old owners. To avoid this case, nodes maintain a list of map entries that need to be moved, and start a separate task that sends map entries to new map owners. When the list is empty, it means that the node has moved all of its map entries, and therefore it broadcasts a message informing the other that this process has finished. When nodes receive such a message from every other non crashed node, the reconfiguration process is terminated.

Reconfiguration can be done on the fly at the expense of performance:

- Every node empties its cache. Executing a get operations will be expensive until the caches are again filled.
- The first remote get operation for some key k requires two more message delays so that the new map owner can fetch the map entry for k remotely.
- Map entries need to be broadcast so that map owners update their maps.

Chapter 3

Implementation

3.1 The dsmDB prototype

To evaluate the dsmDB approach described in Chapter 2, we implemented a prototype of it. In this chapter we add some considerations of practical interest. Our prototype closely follows the system architecture we gave in Section 2.2, and was implemented using the C language. Figure 3.1 shows the architecture of the dsmDB prototype; it adds some details to Figure 2.2.

Applications that use the dsmDB are linked to a client library that conveniently allows to interact with a dsmDB node. The client library implements functions for reading and writing data items, and committing transactions. The Transaction Manager of the dsmDB node handles connections to user applications and their active transactions. To do so, it uses the Transaction module that allows to create transactions and keeps track of read and write sets. Transactions use the Storage Manager for reading data items from Storage or remotely using the Remote module. The Remote interface allows to send read operations to the nodes responsible for a given key; this is done according to the Distributed Mapping mechanism as explained in Section 2.4. Remote modules of different nodes are connected through TPC/IP links.

At commit time, transactions are handed to the Broadcast Manager, which in turn sends them for validation to the Consistency Manager through a reliable TCP/IP link. Transactions at the Consistency Manager are first validated and then broadcast to all the other nodes. Broadcast primitives are implemented on top of the Paxos algorithm [Lam01]. The Paxos protocol is described by the actions taken by different processes, each of which having a particular role in the protocol. In our application of the protocol, the Consistency Manager is a *proposer* and dsmDB nodes are *learners*; between *proposers* and *learners* there is a number of acceptors. Our Paxos protocol is implemented on top of IP multi-

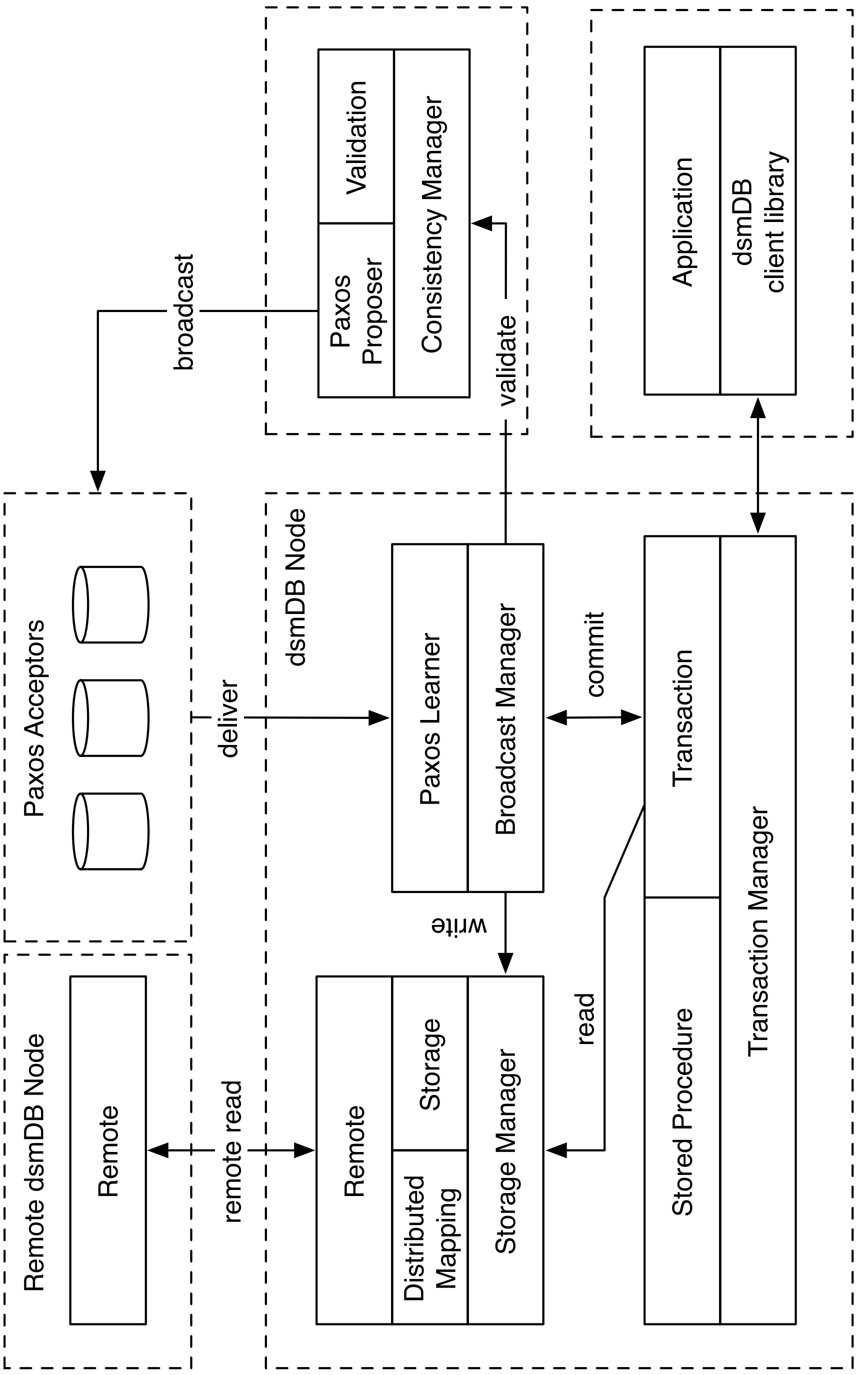

Figure 3.1: Structure of the dsmDB prototype implementation

cast. Transactions broadcast by the *proposer* are delivered in *total order* by the *learners* of each node. When receiving transactions from its *learner*, the Broadcast Manager updates the storage using the write set of committing transactions. The outcome of a transaction, either commit or abort, is propagated back to the application that issued the transaction through the Transaction Manager and the client library.

In the rest of this chapter we give an overview of the Paxos protocol; we explain how data items are stored in our Storage; how to efficiently implement the validation test at the Consistency Manager; and we quickly introduce Stored Procedures.

3.2 Paxos

Paxos is an algorithm for building fault-tolerant distributed systems; it is a protocol for solving *consensus* in a collection of processes. Processes propose values, and the *consensus* algorithm chooses one of these values. When a value is chosen, processes can learn it.

3.2.1 Protocol

The protocol can be explained by the actions taken by three different kinds of processes or roles, namely: *proposers*, *acceptors*, and *learners*. The consensus algorithm proceeds in two phases:

Phase 1 a) A proposer sends a *prepare* message to acceptors containing a ballot number b.
b) If an acceptor receives a *prepare* message with ballot number b, it promises not to accept new proposals less than b, if b is the largest proposal received so far. If the acceptor already accepted a value, then its *promise* message contains also the accepted proposal.

Phase 2 a) If a proposer received a *promise* message in response to its *prepare* message with ballot b from a majority of acceptors, then it sends an *accept* message to the acceptors containing ballot b and a proposal value. If there is at least one acceptor that already accepted some value v, then the proposal value remains v (or one among the set of values that were already accepted), otherwise the *proposer* its own value.
b) If an acceptor received an *accept* message with ballot b and proposal value v, it sends an *accepted* message with ballot b and value v, unless it already replied to a *prepare* message with ballot $b' > b$.

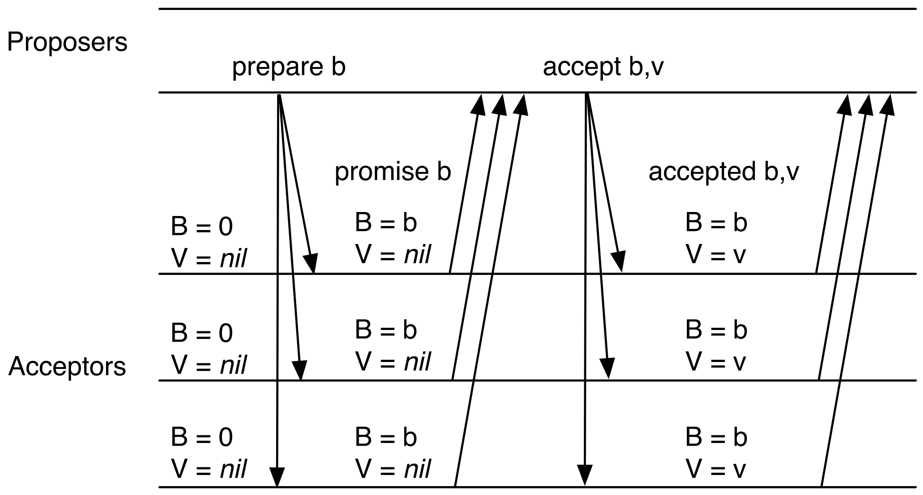

Figure 3.2: Paxos message flow under normal operation.

Figure 3.2 shows the normal message flow between *acceptors* and *proposers* in the Paxos algorithm. *Learners* can learn value v with ballot b, if and only if a majority of acceptors have sent an *accepted* message with ballot b and value v.

3.2.2 Termination

Notice that the algorithm, as described so far, does not ensure *termination*. For instance, consider the example shown in Figure 3.3. A *proposer* sends a *prepare* message for ballot $b1$ to the *acceptors*; follows a *promise* message for ballot $b1$ from the acceptors. Similarly, another *proposer* sends a *prepare* message with ballot $b2 > b1$, and the *acceptors* promise not to accept values with ballots smaller than $b2$. As soon as the first *proposer* tries to send an *accept* message with ballot $b1$, this will be ignored by the *acceptors* because they already promised to reject any value with ballot smaller than $b2$. This behavior can possibly be repeated over and over; it follows that *termination* is not guaranteed.

Typically, the solution to this problem involves a leader election mechanism. Using such a mechanism, a leader is elected among all the *proposer* processes. The leader is the only process allowed to send messages to the *acceptors*. Non-leader proposers, propose their values by sending them to the leader. The leader executes the protocol for its values and the values received from the other proposers.

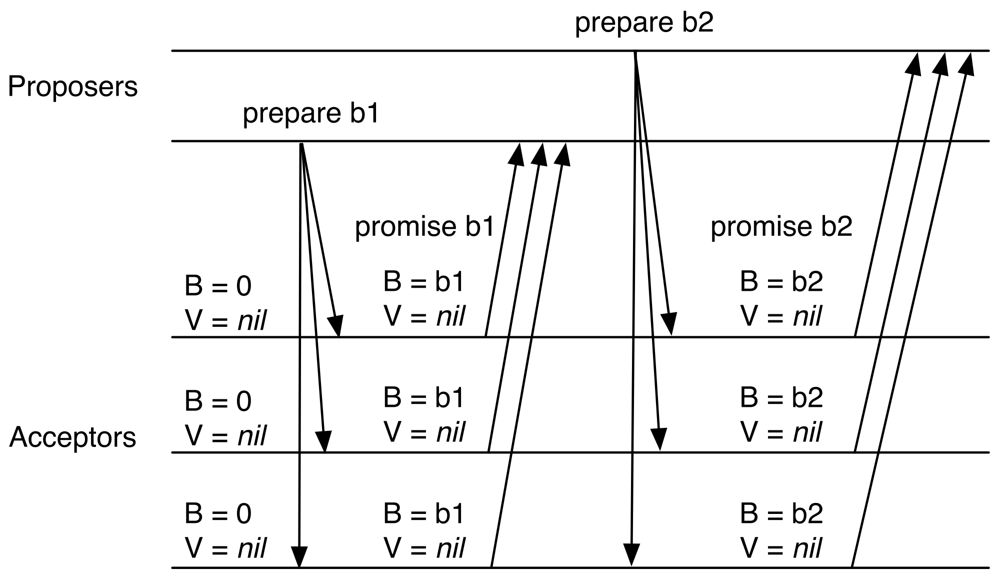

Figure 3.3: Paxos message flow that might never terminate.

3.2.3 Paxos and the dsmDB

Total order broadcast can be achieved by running multiple instances of the protocol we just described. The protocol works in an asynchronous setting where messages can be lost, duplicated or take arbitrary time to be delivered. The protocol works as long as a majority of acceptors is non-faulty. In case of failure, *acceptors* can restart if their state is saved to stable storage, thus before replying to *prepare* and *accept* messages acceptors need to update their state and save it to stable storage.

In our current implementation of Paxos, we assume that there is always one single proposer that never fails (i.e., the Consistency Manager). As an optimization, our implementation also performs *ballot reservation*: instead of sending *prepare* and *promise* messages for every instance the protocol, a proposer can pre-execute phase 1 for several future instances, thus reducing the number of communications delays to two before a value can be learned.

Acceptors use Berkeley DB[1] as a stable storage layer. The Berkeley DB storage is configured not to use any locking mechanisms since the acceptor is the only process that reads and writes its own storage. Since Paxos uses stable storage and the dsmDB uses Paxos to broadcast transaction batches, durability does not depend on the survival of nodes as discussed in Section 2.4, because the state of dsmDB could be recovered from the acceptors.

[1]http://www.oracle.com/database/berkeley-db/index.html

3.3 In-Memory Storage

We explained in Section 2.3 that to increase the level of multiprogramming the dsmDB stores multiple versions of a data item. Moreover, when a node delivers transaction updates, it may decide to cache data items for which it is not responsible.

Figure 3.4 shows the internal data structure we use in our prototype to store both permanent and cached items. We use a regular hash table for storing keys, and collisions in the table are handled using *chaining*: keys that hash to the same position are chained in a linked list of collisions. Every key stored in the hash table is placed in a structure we call *key entry*. Other than the key itself, a key entry contains the head of a versions list, a pointer to the next element in the conflict list, and a pointer to the next element in the lru list.

The list of versions stores all of the available versions of values associated to the key. The list of versions is kept in order, from the oldest to the newest version. We explained in Section 2.3.3 how to limit the number of different versions and what versions can be eliminated. Additionally, we keep an LRU (Least Recently Used) list containing all of the cached keys. Every time a cached key is accessed, it's moved to the front of this list. A garbage collection mechanism periodically deletes cached data items from the tail of the LRU list.

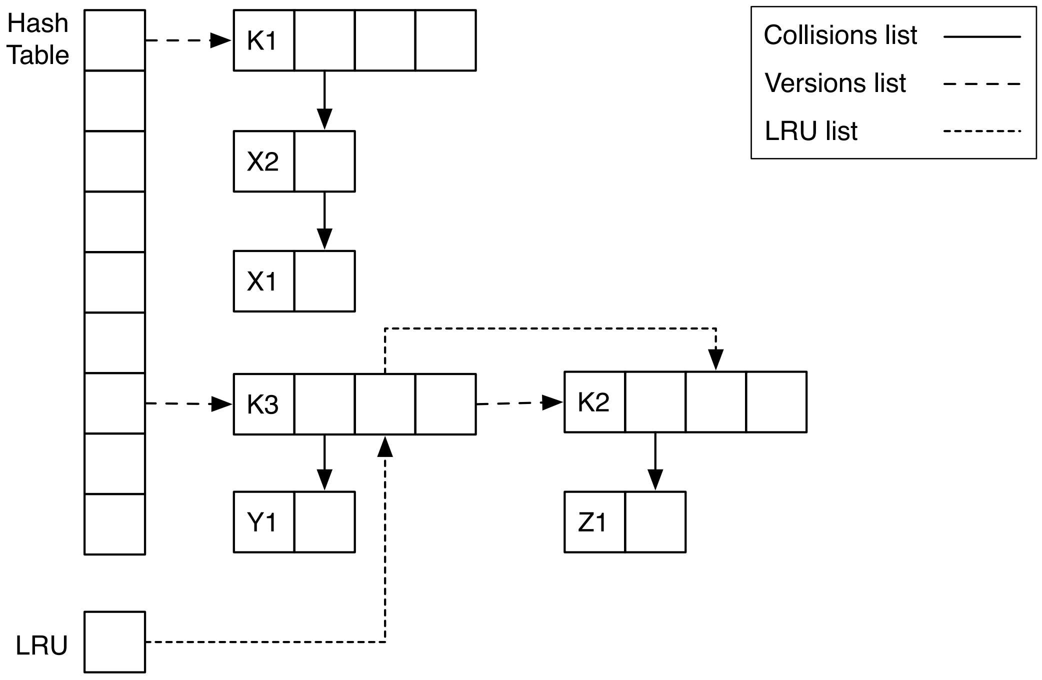

Figure 3.4: Storage data structure

The example in Figure 3.4 shows a hash table with three key entries storing keys $K1$, $K2$, and $K3$. $K1$ has two versions of value X, versions 1 and 2. In this example $K1$ is a permanent item, therefore it is not subject to garbage collection and does not need to be in lru list; whereas $K2$ and $K3$ are cached data items and linked in the lru list.

3.4 Validation test

In this section we explain how to efficiently implement the validation test described in Section 2.3.2. The Consistency Manager holds an array of transactions we called the *commitBuffer*. A transaction T contains a read set $T.rs$ and a write set $T.ws$; these are respectively the set of keys read and the set of keys written by T. A transaction commits if the validation test finds a position i in the *commitBuffer* such that the following two statements hold:

1. For all transactions $T' \in \text{commitBuffer}[1, i-1]$: $T'.ws \cap T.rs = \emptyset$.
2. For all transactions $T' \in \text{commitBuffer}[i, \text{bufferSize}]$: $T'.rs \cap T.ws = \emptyset$.

To find such position, we use algorithm Algorithm 3. The first while loop finds up to which position the condition (1) holds; the position is stored in variable *index*; the second while loop checks whether condition (2) holds; if so T is inserted in *commitBuffer* at position *index*.

We use bloom filters to efficiently implement the intersection operation between two sets. Checking whether the intersection of two bloom filters is empty is a matter of ANDing their bitmaps, and checking that the resulting bitmap contains only zeros. Notice that, this can be done only if both bloom filters have the same size. Using bloom filters offers two advantages: (1) finding the intersection of two bloom filters is linear in the size of their bitmap; and (2) the information stored in bloom filters is compressed, thus reducing the memory requirements. However, bloom filters have the disadvantage of false positives: a small portion of transactions is aborted due to false positives.

3.5 Stored Procedures

In our current implementation of the dsmDB, the client library communicates with a single dsmDB node running on the same machine using Unix Pipes. Every get and put operation of a transaction needs to be sent from the client library to the node over pipes. DsmDB nodes process one client operation at a time. Stored

Algorithm 3: Validation test

```
function Validate(T)
    i ← 0
    while $i < bufferSize$ do
        $T'$ ← commitBuffer[i]
        if $T'.ws \cap T.ws = \emptyset$ then
            $i \leftarrow i + 1$
        else
            break
    index ← i
    while $i < bufferSize$ do
        $T'$ ← commitBuffer[i]
        if $T'.rs \cap T.ws = \emptyset$ then
            $i \leftarrow i + 1$
        else
            break
    if $i = bufferSize$ then
        insert(T, commitBuffer, index)
```

procedures are used to avoid the cost of several round trips for transactions that are executed frequently. The code of a transaction can be embedded into the dsmDB node. Stored procedures require only one round trip over pipes: the client library sends the invocation of a stored procedure along with its arguments to the dsmDB node; the dsmDB node executes the stored procedure with the given arguments and replies with the result to the client.

Stored procedures are implemented on top of the Transaction module, which handles the internal representation of a transaction and gives a convenient API that implements get, put, and commit operations.

The dsmDB prototype currently lacks a mechanism to “install” or “plugin” new stored procedure types in a flexible way. This can be achieved in different ways, for example by using dynamically loadable libraries. Stored procedures are compiled into dynamic library, and dsmDB nodes load them whenever needed.

Chapter 4

Performance Evaluation

4.1 Infrastructure

The experiments in this chapter are performed on a cluster of 16 Apple Xserve G5 machines, with 2 G5 CPUs of 2.3 GHz, between 1 and 2 GB of main memory, and gigabit network links. The operating system used is MAC OS X 10.4.11.

4.2 Benchmarks

In this section, we describe two benchmarks that we used to evaluate the performance of the dsmDB: the standard TPC-B [TPC94] benchmark, and a distributed B-tree implementation on top of the dsmDB.

4.2.1 TPC-B benchmark

Although TPC-B is considered an obsolete benchmark [LGK93], we decided to implement it because it would be difficult to implement newer benchmarks such as TPC-C. The main criticism against TPC-B is that it represents a simplistic benchmark because it does not model the requirements of real OLTP systems.

Schema

TPC-B models a simple banking application composed of *branches*, *tellers*, *accounts* and the *history* of the executed transactions. Figure 4.1 shows that the database is composed of 4 different tables. Each table maintains a balance for each branch, teller, and account record in the system. A branch has a one-to-many relationship with the tellers and the accounts. For each executed transac-

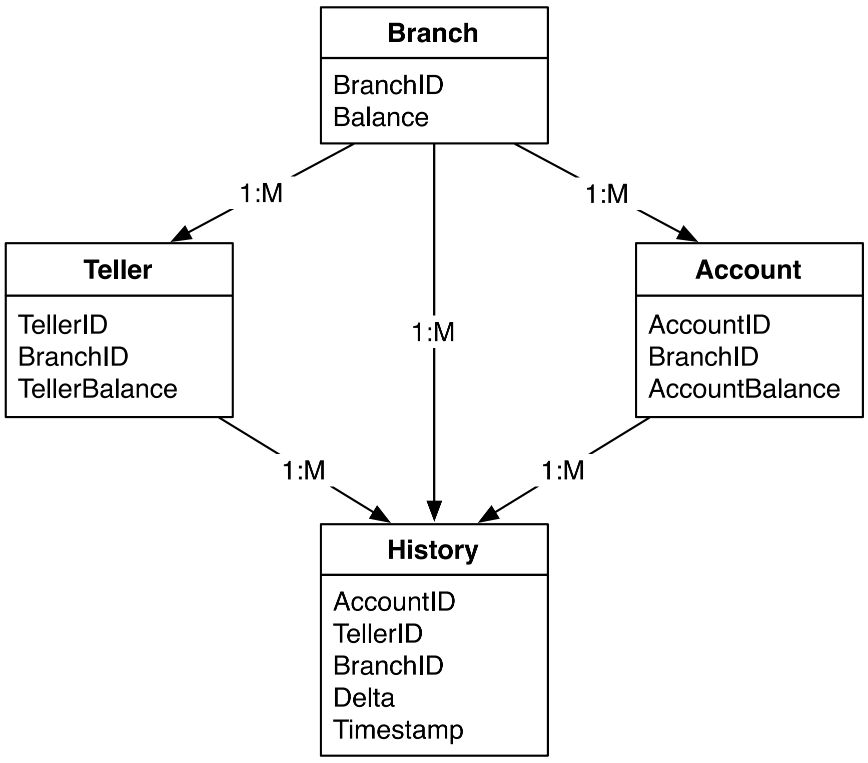

Figure 4.1: TPC-B schema layout.

tion the benchmark creates a history record that keeps track of the amount of money that was withdrawn or deposited in an account. Changing the balance of an account means that also the balance of the corresponding teller and branch must be changed. Account, teller, and branch records must contain at least 100 bytes; history records must contain at least 50 bytes.

Transactions

TPC-B defines a single transaction that performs the following statements:

```
BEGIN TRANSACTION
    Update Account where Account_ID = Aid:
        Read Account_Balance from Account
        Set Account_Balance = Account_Balance + Delta
        Write Account_Balance to Account
    Write to History:
        Aid, Tid, Bid, Delta, Time_stamp
    Update Teller where Teller_ID = Tid:
        Set Teller_Balance = Teller_Balance + Delta
        Write Teller_Balance to Teller
    Update Branch where Branch_ID = Bid:
        Set Branch_Balance = Branch_Balance + Delta
        Write Branch_Balance to Branch
COMMIT TRANSACTION
Return Account_Balance
```

In terms of dsmDB transactions, this can be translated into 3 get operations to retrieve the given account, teller and branch records; and 3 put operations to update these records; an additional put operation to create a new history record. A total of 7 operations. For this benchmark type, transactions are submitted in batch mode, that is clients submit one transaction at a time as fast as they can; there is no “think time” between consecutive transactions.

Scaling rules

The TPC-B specification also defines standard *scaling rules*. The benchmark must contain 100’000 accounts, 10 tellers and 1 branch. These numbers can be changed with the constraint that they are all changed proportionally. For example, if one wants to double the number of branches, then also the number of accounts and tellers must be doubled.

Unfortunately, this requirement is not feasible for the dsmDB. Recall the validation test described in Section 2.2.5: if two transactions read a data item and at least one them also updates it, then one of the two transactions must be aborted. We would experience a very high abort rate and no concurrency at all. Having one single branch is therefore not feasible, and by increasing the number of branches to a meaningful number we would need to have a large number of accounts, which is also not very practical. We therefore decided to redefine our scaling rules as follows: 100 branches, 1'000 tellers and 100'000 accounts.

Measurements

In TPC-B, clients run in parallel for a fixed amount of time, and at the end of the benchmark execution the following measurements are reported:

Number of executed transactions: The total number of transactions that were executed.

Number of committed transactions: The portion of executed transactions that were reported to be committed.

Number of aborted transactions: The portion of executed transactions that were reported to be aborted.

Average residence time: Average time needed to execute a transaction. Residence time is defined as $RT = t_2 - t_1$, where t_1 is a timestamp taken by the client before executing the first get operation that started the transaction, and t_2 is a time stamp taken after receiving the result of the transaction (either committed or aborted).

TPS: Committed transactions per second, computed as Number of committed transactions / Benchmark execution time.

4.2.2 B-tree benchmark

To overcome the limitations of the TPC-B benchmark, we designed a benchmark that implements searches and inserts in a B-tree. It is interesting to implement and measure the performance of a B-tree on top of the dsmDB because:

- It has, as we will see, interesting properties that can be exploited by dsmDB's caching and concurrency control mechanism.
- It can be used as a "cheap" way for extending dsmDB's query model. For instance, range queries can be implemented on top of B-trees.
- It serves as a use-case and excellent example of building distributed data structures on top of the dsmDB. Because of dsmDB's shared memory programming model, distributed data structures can be easily implemented.

Properties of B-trees

A B-tree (complete description and algorithms can be found in [CSRL01]) is a balanced search tree optimized for storing and retrieving data from secondary storage. The idea is that every node maintains multiple keys and multiple children. Usually the number of children or "branching factor" is optimized so that a node fits in a disk page, reducing the height of the B-tree and therefore reducing the number of disk I/O operations when searching or inserting elements in the tree. This applies also in the context of the dsmDB, the number of remote lookups (which is a relatively expensive operation to perform) is low. A B-tree is kept balanced so that its minimum degree t is subject to the following restrictions:

1. Every node of a B-tree, except the root, has at least $t-1$ keys.
2. Every node of a B-tree, contains at most $2t-1$ keys.

Insertion and removal of keys is done so that the above properties always hold. Insertion of a new key in a node that already holds $2t-1$ keys causes the node to be split; similarly, the removal of a key from a node that contains $t-1$ keys causes a merge. The height of a B-tree is $O(\log_t(n))$ where n is the number of items in the B-tree.

Transactions

Our B-tree implementation supports two operations:

Btree-Search(k): this operation returns the value associated to key k.

Btree-Insert(k, v): this operation stores the key-value pair (k, v) in the tree.

In terms of dsmDB transactions, both operations require at most $\log_t(n)$ reads. A `Btree-Search` operation starts from the root and traverses the inner nodes of the tree until it finds the node that stores the value with the given key. Similarly a `Btree-Insert` starts from the root and traverses the inner nodes of tree until it finds the node in which to store its given value. While traversing, a `Btree-Insert` operation ensures the balancing property of the tree by splitting nodes if necessary.

Notice that the inner nodes of a tree are the most accessed data items. Every dsmDB node with sufficient cache space will automatically cache inner nodes in its storage, thus further reducing the number of remote lookups. Moreover, contention is extremely low if we assume that `Btree-Insert` operations are performed over randomly chosen keys (i.e., if inserts are not performed in the order in which nodes are indexed).

4.3 Experiments and Results

4.3.1 DsmDB vs. Berkeley DB

In this experiment we compare the dsmDB against Berkeley DB.[1] Berkeley DB is a standalone database that can be embedded into applications; it is widely used in industry, and in open source projects. Its design is inherited from traditional databases: it provides locking and logging mechanisms. It is similar to the dsmDB when it comes to its interface: it provides a key-value interface, and completely avoids SQL or any high level query languages. In this experiment we compare dsmDB's TPC-B benchmark against BDB's implementation of the same benchmark.

Experiment setup

Since Berkeley DB is a standalone database, we limit the dsmDB to run in a single node. We also run 3 acceptors and the Consistency Manager. The Berkeley DB storage layer in the acceptors, is setup as a data store and does not need locking because only one process accesses the database at the time. Moreover, the acceptors are configured so that disk writes are performed asynchronously.

The Berkeley DB benchmark needs to use the transactional interface, therefore logging and locking is required. To make the results comparable, the logging subsystem is configured so that disk writes are performed asynchronously.

We first run the dsmDB and fix the duration of the benchmark to 1 minute. We then execute the same number of transactions on Berkeley DB. In the results we compare both throughput and abort rate. The benchmark is repeated for 1, 2, 4, and 8 concurrent number of processes accessing the database.

Experiment results

Figure 4.2 shows the throughput measured in transactions per second over an increasing level of concurrency. The graph shows that the performance characteristics of Berkeley DB and the dsmDB are very different. Berkeley DB performs almost 3 times better than the dsmDB in the case of 1 client accessing the database; whereas the dsmDB performs much better in the case of concurrent clients accessing the data.

There are several factors that explain these differences. In Berkeley DB, disk becomes immediately the bottleneck as soon as there are two concurrent clients

[1] http://www.oracle.com/database/berkeley-db/index.html

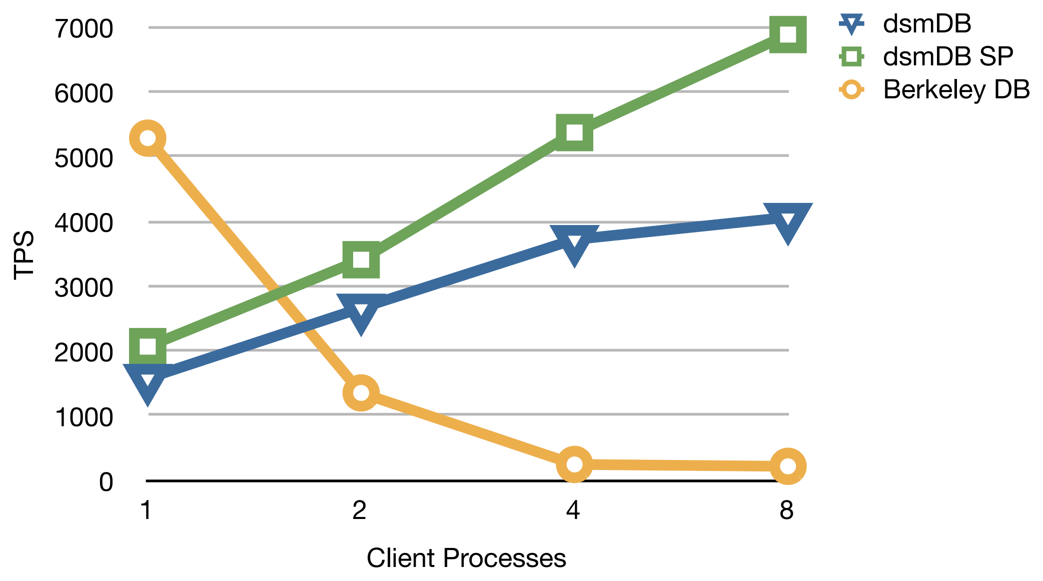

Figure 4.2: Throughput comparison between Berkeley DB and dsmDB.

accessing it. Each transaction executing in Berkeley DB potentially requires several filesystem operations: seeking to the database file and reading it; seeking to the log file and writing it. In fact, the typical way to optimize throughput in BDB is to place log and database files on two different disks.

A dsmDB transaction is first executed in memory, sent to the Consistency Manager for validation where it gets batched together with other concurrent transactions, and finally, the transaction batch is written on stable storage in the acceptors. Acceptors use Berkeley DB as storage layer without transactional guarantees; no locking nor logging is required in the acceptors. The number of filesystem operations is reduced because: (1) transactions are batched and written to disk in sequential order; and (2) there is no log file to write.

Figure 4.3 shows abort rates reported after the same benchmark execution. This shows that another limiting factor is the locking mechanism. The lock granularity of Berkeley DB is at page-level. To increase I/O performance, the page size is typically the same as the size of a disk block. Locking can be a considerable overhead as transactions are synchronized; transactions might be aborted because of deadlocks or timeouts when acquiring locks.

DsmDB nodes perform one operation (get, put or commit) at a time. Thus completely avoiding locking and deadlock detection mechanisms. The concurrency control mechanism, as explained in Chapter 2, works over multiple versions of data items and is optimistic.

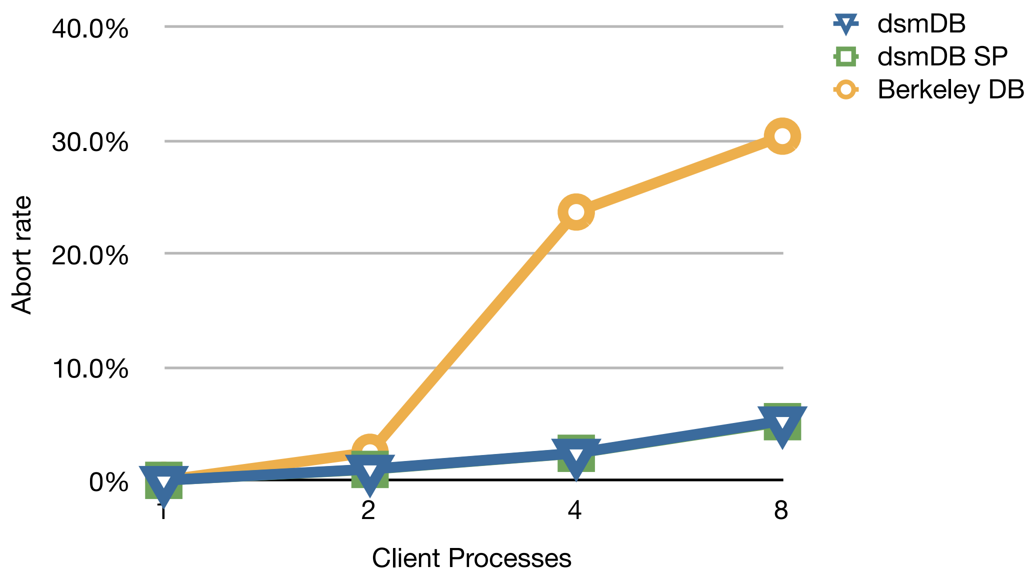

Figure 4.3: Abort rate comparison between Berkeley DB and dsmDB.

4.3.2 TPC-B benchmark with multiple nodes

In this benchmark we evaluate the TPC-B benchmark over multiple dsmDB nodes. We run the TPC-B over an increasing number of clients and increasing number of nodes.

Experiment setup

We run the TPC-B benchmark over 2, 4, and 8 nodes and repeat the experiments for 1, 2, 4 and 8 clients per node. Clients first connect to the their dsmDB node and simultaneously start submitting transactions for a total of 4 minutes. In each run the length of the commit buffer in the Consistency Manager is set equal to the number of nodes. The replication factor is set to 2 — each data item is replicated at 2 two distinct nodes.

Experiment results

Figure 4.4 shows the throughput in the TPC-B benchmark for all of the configurations we considered. In particular each line represents committed transactions per second over an increasing number of clients per node. In general we observe that by increasing the number of clients, throughput saturates faster on the configuration with a low number of nodes. Although throughput improves as we increase the number of nodes, we do not observe perfect scalability (i.e., throughput does not double when we double the number of nodes). There are many reasons that explain the performance behavior shown in this benchmark:

- A 2-node configuration starts with an advantage: since the replication factor is 2, the data set is fully replicated and nodes never need to fetch items remotely.
- As we will show, the workload of the TPC-B benchmark has high contention. Moreover, it is a write-intensive benchmark: every transaction performs 4 writes, 3 updates, and 1 append.

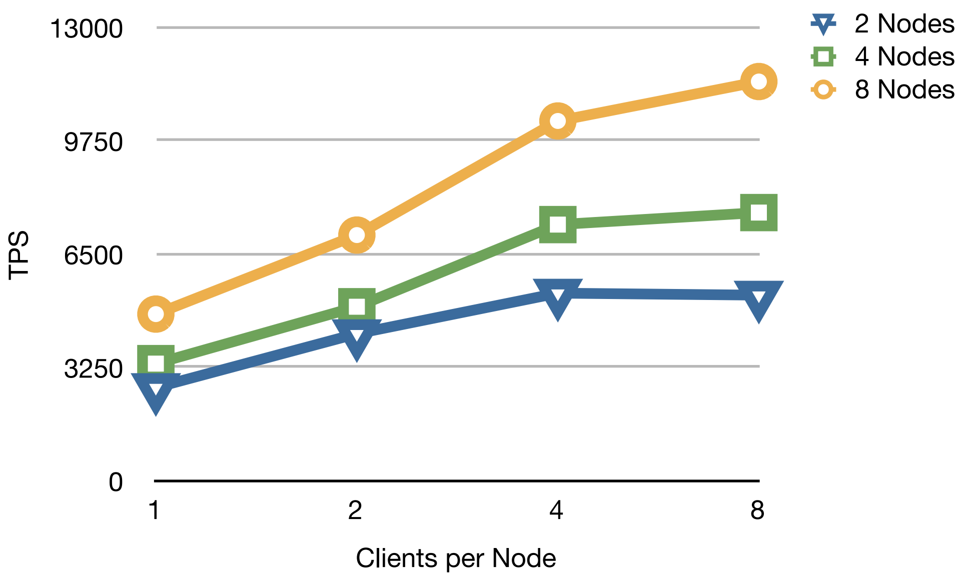

Figure 4.4: Throughput in the TPC-B benchmark over multiple nodes.

Recall from Section 4.2.1 that there are 100 branch records in the TPC-B benchmark, and that before executing a transaction clients choose a branch record randomly. Consider the 8 nodes configuration running 8 clients per node. In this case, there is a total of 64 clients running in parallel, therefore it is very likely that distinct clients choose the same branch record. In fact, as shown in Figure 4.5, as we increase the number of clients we observe that abort rates increases rapidly.

Figure 4.6 shows latency versus throughput for each configuration. To give a better idea of how the system scales, we select those configurations that bring

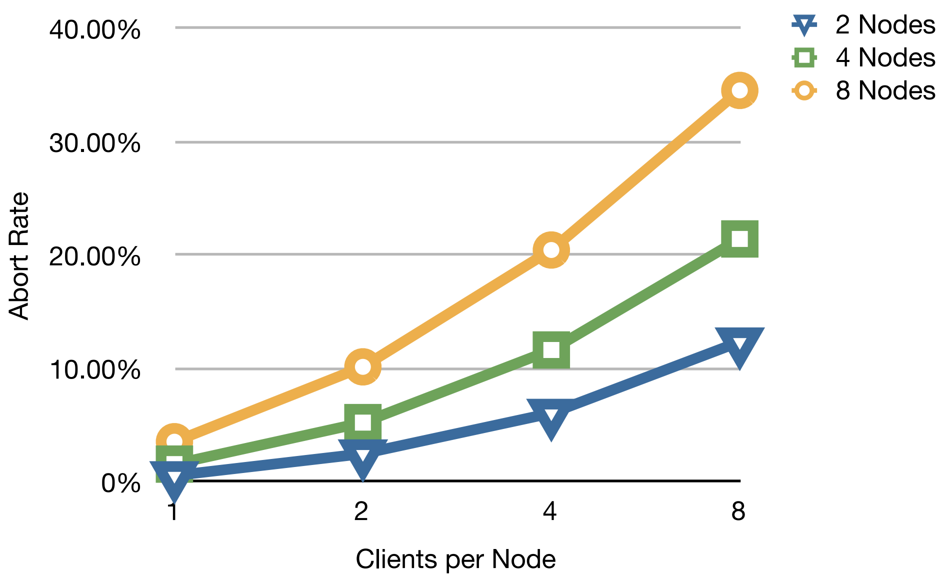

Figure 4.5: Abort rate in the TPC-B benchmark over multiple nodes.

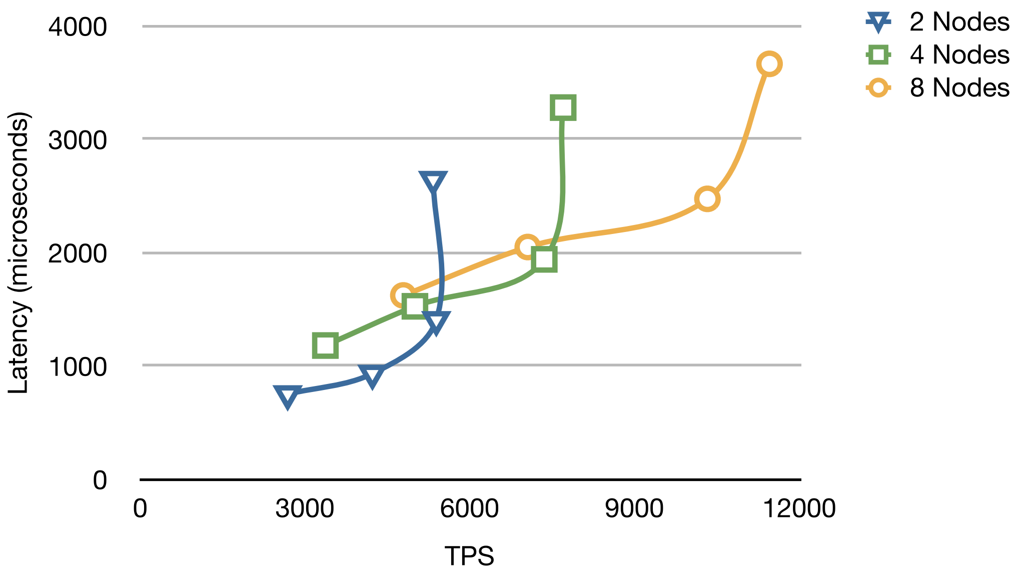

Figure 4.6: Latency versus throughput in the TPC-B benchmark over multiple nodes.

the system at peak load but do not saturate the system (i.e., increasing the number of clients, increases latency but not throughput). Scalability at peak load is shown in Figure 4.7. In particular it compares: (1) throughput as we have measured it in the 2, 4, and 8 nodes configurations (TPS); (2) throughput as if all transactions would commit, same workload without contention (TPS no aborts); and (3) throughput as if the system would scale linearly a linear line showing what our system should achieve if it was perfectly scalable (TPS perfect scalability).

Figure 4.7 shows that contention in the workload is indeed a factor that affects scalability, but it is not the only one. We attribute the scalability gap also to the performance of our Paxos implementation. Recall that the length of the commit buffer in the Consistency Manager is set to the number of nodes. Therefore, as we increase number of nodes, the commit buffer increases too, and since the workload is write-intensive, every transaction must be broadcast. Table 4.1 summarizes the throughput, measured in delivered values per second, of our Paxos implementation.[2] For instance, consider an 8-node configuration with 4 clients per node. In such a configuration, the commit buffer is sent through Paxos when it contains around 4000 bytes (8 transactions). From Table 4.1 we get that Paxos can deliver at most $8 * 1800 = 14400$ transactions per second. This means that an 8-node configuration reaches the limit of our Paxos library. Write-intensive workloads are easily limited by the performance of Paxos.

Value Size (bytes)	Values / sec
1000	3800
2000	2300
4000	1800
8000	1300

Table 4.1: Delivery rate of Paxos for different value sizes.

To conclude this experiment, we show in Figure 4.8 the throughput in the TPC-B benchmark using Stored Procedures. By using Stored Procedures the throughput increases up to 10%. The performance characteristics remain the same, with the only difference that the system saturates “earlier”, with fewer clients.

[2]Marco Primi, *LibPaxos Performance Analysis*, http://libpaxos.sourceforge.net/files/Primim-SPLab08.pdf

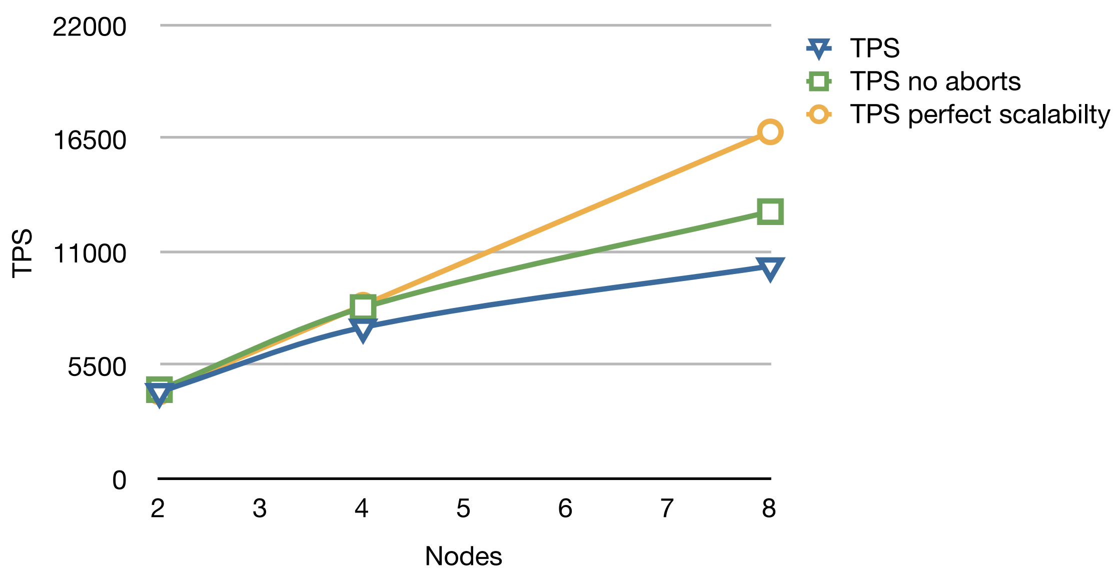

Figure 4.7: Scalability in the TPC-B benchmark over multiple nodes.

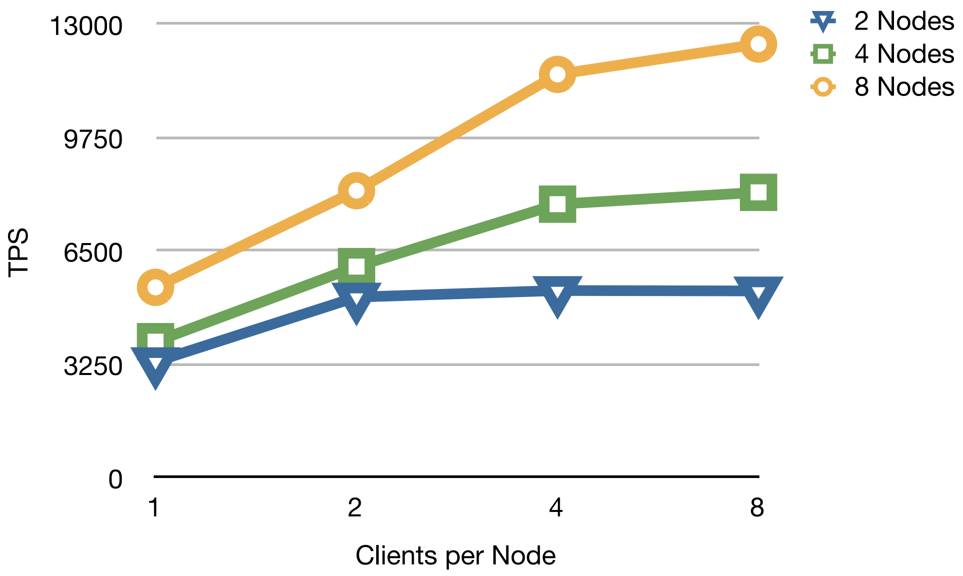

Figure 4.8: Throughput using Stored Procedures in the TPC-B benchmark over multiple nodes.

4.3.3 B-tree benchmark results

In this experiment we evaluate our B-tree on top of the dsmDB implementation. The workload contains 50% B-tree insert operations and 50% B-tree search operations. As opposed to the TPC-B benchmark, the B-tree benchmark is not write intensive, as half of the transactions are read-only.

Experiment setup

The B-tree benchmark consists of executing 2'000'000 operations, 1'000'000 inserts and 1'000'000 lookups. We repeat the experiment for 4, 6, and 8 nodes. We fixed the number of clients per node to 6 (we determined that 6 is the number of clients per node before saturation).

Clients using the B-tree insert fixed-size key-value pairs. We use 8 bytes for the key and 100 bytes for the value. The minimum degree of the inner nodes of the B-tree is 12, therefore each node holds at most $(2 \cdot 12) - 1 = 23$ "pointers" to values. The size of an inner node is around 500 bytes. At the end of the benchmark the size of the database is $(1000000 \cdot 100 bytes) + ((1000000/24) \cdot 500 bytes) = 115 MB$. The replication factor is set to 2, therefore at the end of the benchmark the system stores around $230MB$ of data. Every node is limited to use at most 100 MB of memory thus, limiting the number of cached items at each node. At the end of the benchmark, the height of the B-tree is $\log_t(n) = \log_{12}(1000000) \approx 5.5$. In the worst case a lookup requires between 5 and 6 reads, and the worst case insert requires at most 5 to 6 reads and writes (multiple writes are required if multiple nodes need to be split).

Experiment results

Figure 4.9 shows the number of operations per second measured on 4, 6, and 8 nodes. As opposed to the TPC-B benchmark, results gathered on the B-tree benchmark are more linear with respect to the number of nodes. The dsmDB shows a scalable behavior (at least up to 8 nodes) because of: (1) low contention, and (2) the presence of read-only transactions.

As Figure 4.10 demonstrates, we measured abort rates under 1% on all of the configurations we considered. Scalability comes also from the fact that half of the transactions are read-only. These kind of transactions always commit, and need not be sent through Paxos. Moreover, since inner-nodes are the most frequently accessed items of the tree, execution is sped up when there is enough memory for caching the inner-nodes.

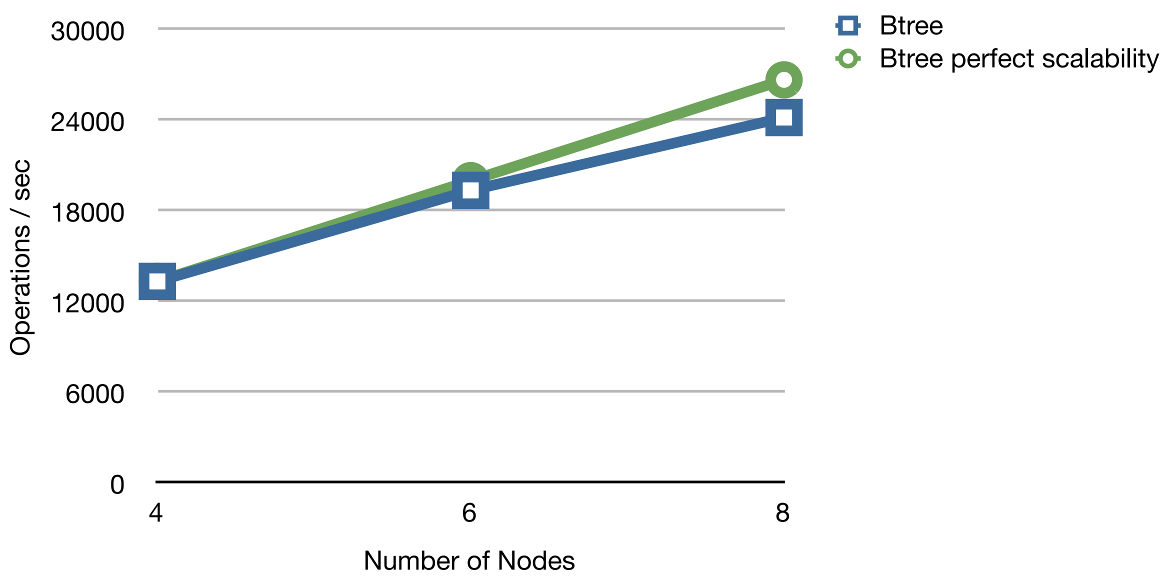

Figure 4.9: Throughput of the B-tree measured in operations per second.

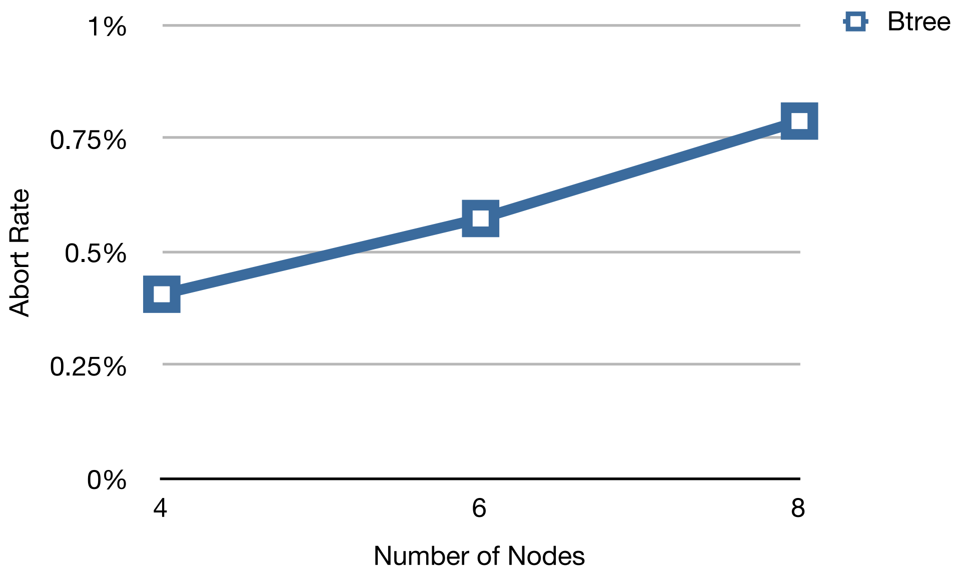

Figure 4.10: Abort rates in the B-tree benchmark.

Chapter 5

Conclusions

5.1 Summary and discussion

We discussed in Chapter 1 several issues in traditional database management systems. Our impression is that performance in these systems is limited by those mechanisms that avoid “bad things” to happen. Moreover, these systems were designed to perform on centralized architectures. These considerations motivate the study of distributed database architectures.

In Chapter 2 we discussed the dsmDB approach. The dsmDB is a distributed system in which each node shares a portion of its memory. Large data sets are stored in the aggregated main-memory of a cluster of machines. The dsmDB eliminates the need of traditional expensive mechanisms like database images on disk, and pessimistic concurrency control mechanisms. Transactions in the dsmDB are first executed over an in-memory storage layer that guarantees only weak consistency, and then globally validated to ensure stronger consistency. We also described a procedure for recovering the state of crashed nodes from the state of non crashed nodes. Similarly, we described a procedure that allows incremental expansion of the system.

In Chapter 3 we described some of the implementation details of our dsmDB prototype. We described the Paxos algorithm that provides our fault-tolerant total order broadcast primitives. Paxos is the only component of the dsmDB that relies on stable storage. But it makes efficient use of disk: it writes batched transaction updates sequentially.

In Chapter 4 we evaluated the performance of the dsmDB. We analyzed the performance over two distinct benchmarks: an implementation of the standard TPC-B benchmark and an implementation of a distributed B-tree on top the dsmDB.

We compared a 1 node dsmDB configuration against the performance of

Berkeley DB, an embedded database extensively used in practice. We found that Berkeley DB performs better when only 1 process accesses the database at a time. Mechanisms such as logging, synchronization, and deadlock detection become problematic as soon as two processes access the database concurrently. We have also shown that the performance of a single dsmDB node is comparable to the performance of Berkeley DB.

We also ran the TPC-B benchmark over multiple dsmDB nodes. We found that scalability is limited by both contention and the write-intensive nature of the TPC-B benchmark. This benchmark represents a worst-case scenario for the dsmDB. Since every transaction updates the state of database, every single transaction has to be broadcast.

The dsmDB performs much better in the B-tree benchmark. In this benchmark the dsmDB has shown a scalable behavior due to the presence of read-only transactions. Read-only transactions in the dsmDB are optimized because they need not be broadcast at commit time and never abort.

5.2 Future work

A number of areas, of both theoretical and practical interest, need future work. We identify the following research lines:

- An extended version of the dsmDB approach that allows deployment on multiple and geographically distributed clusters. Broadcasting primitives should be replaced by multicast primitives, and the protocol should be augmented to guarantee high-availability despite crashes or outages of entire clusters.

- Investigate other consistency criteria that can be implemented on top of a distributed shared memory mechanism that provides only weak level of consistency.

- A proof of correctness of the concurrency control mechanism given in Section 2.3. Such a proof would demonstrate that the protocol meets the definition of one-copy serializability and allows to execute only schedules that meet the definition.

The dsmDB prototype presented in Chapter 3 can be improved in a number of ways:

- We have seen that scalability is an issue when it comes to write-intensive workloads. The performance problem can be reduce as discussed in [Pri09].

- Recovery and Reconfiguration procedures have not been implemented in our prototype.
- Data sets in the dsmDB are stored in a single domain. Multiple domains would allow to perform `get` and `put` operations on distinct tables.
- The query interface of the dsmDB can be augmented with range queries. For instance, we already mentioned one way to address this issue in Section 4.2. Implementing a B-tree (or similar) data structure on top of the dsmDB represents a flexible way to extend the query interface. However, embedding such functionality directly in the dsmDB might lead to better results in both performance and memory overhead for keeping the data structure itself.

Bibliography

[AMS+07] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 159–174, New York, NY, USA, 2007. ACM.

[BHG86] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[BM02] Andrei Broder and Michael Mitzenmacher. Network applications of bloom filters: A survey. In *Internet Mathematics*, pages 636–646, 2002.

[CSRL01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[FP08] Nelson Duarte Filho and Fernando Pedone. dsmdb: a distributed shared memory approach for building replicated database systems. In *SDDDM '08: Proceedings of the 2nd workshop on Dependable distributed data management*, pages 11–14, New York, NY, USA, 2008. ACM.

[HJK+07] Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *In Proc. SOSP*, pages 205–220, 2007.

[Lam01] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, December 2001.

[LGK93] Charles Levine, Jim Gray, and Walt Kohler. The evolution of tpc benchmarks: Why tpc-a and tpc-b are obsolete, 1993.

[NL91] Bill Nitzberg and Virginia Lo. Distributed shared memory: A survey of issues and algorithms. *Computer*, 24(8):52–60, 1991.

[PGS03] Fernando Pedone, Rachid Guerraoui, and André Schiper. The database state machine approach. *Distrib. Parallel Databases*, 14(1):71–98, 2003.

[Pri09] Marco Primi. Paxos made code. Master thesis submitted to the Faculty of Informatics of the University of Lugano, 2009.

[SMA$^+$07] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 1150–1160. VLDB Endowment, 2007.

[TPC94] TPC. TPC Benchmark B: Standard specification (revision 2.0), 1994.