Scalable distributed data structures for database management
Karlsson, S.J.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 6

Scalable Distributed Database Storage Manager/Database System

This chapter essentially contains material from the paper "Transparent Distribution in a Storage Manager" [KK98] presented at PDPTA'98 in Las Vegas.

We explore one direction of implementation of an SDDS-enabled database system. The system is based on MONET [BK95] main-memory database system and allows for a seamless integration.

6.1 Seamless SDDS integration in an Extensible DBMS

Abstract

We will now validate the concept of integrating a Scalable Distributed Data Structure (SDDS) into an extensible DBMS. Seamlessly allowing for parallel processing in a multicomputer. We show that this merge provides high performance processing and scalable storage of very large sets of distributed data.

Further on, we show by extending the relational algebra interpreter that access to data, whether it is distributed or locally stored, can be made transparent to the database application (user). This concept potentially allows existing applications of database systems to efficiently process much more data than a single workstation.

By seamless integration in the extensible database system we have brought laborative/isolated studies on SDDSs into the realm of viable alternatives for
distributed database systems.

We illustrate the performance efficiency by several experiments on a large network of workstations. For several operators we achieve perfect scale-up, i.e. doubling the number of nodes allows double amount of processing in the same time.

6.2 Introduction

Over the last decades major progress has taken place in efficient data management in a distributed setting. Many commercial DBMS already provide the hooks to control data distribution data and query optimizers to exploit it. However, modern applications, such as GIS and Data Mining, continue to stress the need for better techniques; both in terms of its scalability and its performance. Especially the limited and rigid scheme deployed by the data distributions and the resulting complexity of the query optimizers hinder major breakthroughs.

In this paper we promote deployment of the SDDSs at a broader scale. First, the SDDS LH* [LNS93] has been integrated in a full fledged extensible database system. Second, we demonstrate how the functionality of SDDS can be exploited for improving performance of certain algebraic operations.

Scalable Distributed Data Structures (SDDSs) [LNS93] [LNS96] in particular addresses the issue of scalable storage, i.e. the ability to administer any foreseen and not foreseen amount of data distributed over a number of processing nodes. Most studies have been focused on algorithm and experiments in an isolated context, i.e., not integrated in a database system.

Their integration solves a problem often encountered in distributed systems, where manual data (re-)distribution are required to handle increasingly larger data sets. SDDS storage scale automatically without any costly re-organization of all data in lock-step mode. They indeed provide transparent fragmentation.

Our target experimentation platform is a network multi-computer, i.e. a collection of workstations and SMP (Symmetric Multi Processor) computers. It enables the system to grow over time by adding components to meet the increased storage and processing demands.

The outline of our paper is as follows. In Section 6.3 we shortly describe the key properties of Scalable Distributed Data Structures and the implementation platform Monet. Then, in Section 6.4, we describe the implementation and its integration of SDDSs in Monet. The algebraic operators are analyzed in Section 6.4.4. Section 6.5 shows performance measures of our implementation. Finally, in Section 6.6 we conclude our work and give directions for future research.
6.3 Background

In this section we give a short introduction to the key concepts of Scalable Distributed Data Structures and the Monet database system.

6.3.1 Scalable Distributed Data Structures

Scalable Distributed Data Structures (SDDS) [LNS93] can be classified as a general access-path mechanism, examples are [Dev93] [WBW94] [KW94] [LNS94] [KLR96] [Kar97]. SDDSs allow storage of a very large number of tuples distributed over any number of nodes. The tuples are distributed to different nodes according to their key value and the state of the SDDS. The primary means for retrieval is again the key value. The objective of SDDSs is to minimize the messages needed to locate the tuple anywhere in the system. Typically, a tuple can be accessed with at most two network messages. One message for sending the request of the tuple and one message for returning the data.

SDDSs differs from other distributed schemas in that it allows the number of nodes to increase at very small cost. Many other distributed data structures require a (costly) total re-organization for adding a node, because they employ static distribution schemas. Examples proposed in the literature are round-robin [Cor88], hash-declustering [KTMO84], range-partitioning [DGG+86]. The ability of SDDSs to scale to several nodes by acquiring one node at a time and gradual reorganization opens up new areas of storage capacity and data access.

For accessing data stored using an SDDS, a client can calculate where a tuple resides using the key. In the case that the client is not fully aware of the number of servers involved in storing the distributed data, the server receiving the request will forward it towards its correct destination. The reason that the client’s calculation may lead to addressing errors is that the clients are not actively updated when the SDDS is partly re-organized. The facts on how the data is distributed is called the clients image. When a server receives a mal-addressed request it will in addition to the forwarding send back an Image Adjust Message to the client. This message improves the image of the client, preventing it from repeating the same mistake. The rationale behind this is that the number of clients may be too large to be efficiently updated continuously, also, not all clients might be interested in the most accurate information at all times. This design decision is aimed at minimizing communication overhead. The incurred cost for updating clients has been shown to be low [LNS96] in laboratory settings on a simulation studies.

In conclusion the main three features of an SDDS are:

- There is no central directory that clients have to go through for data access. This avoids hot-spots.
CHAPTER 6. SCALABLE DISTRIBUTED STORAGE MANAGER

- Each client accessing the data, has an approximately image of how the data is distributed. The image is lazily updated by servers. The updates only occurs when clients make addressing errors.

- The servers are responsible to handle all requests from clients, even if the client makes an addressing error. It is also the responsibility of the server to update the client.

LH* [LNS93], which is our choice of implementation, is a distributed variant of Linear Hashing [Lit80]. The LH* schema allows efficient hash-based retrieval of any tuple based on the tuple key. Insertion requires, on average, one message and retrieval two messages. If the clients image is outdated the request is forwarded and the client is updated by the server.

The prime drawback of the earlier reported studies, is that the operations directly supported are limited to individual tuple access. It requires generalization to support a relational algebra, which we address in this paper.

6.3.2 Monet

Monet\(^1\) [BK95][BMK99] provides for the Next Generation DBMS Solutions using todays trends in hardware and operation system technology. Monet's features include: decomposed storage model; using binary relations only; main-memory algorithms for query processing; modular extensibility for data structures; and indices and methods. High throughput is achieved with extensive use of bulk operators.

Monet is successfully used in Data Mining applications[HKMT95] and GIS[BQK96], and its supreme performance has been demonstrated against several benchmarks, including OO7[BKK96] and TPC-D[BWK98].

6.4 SDDS within an Extensible Database System

Adding SDDSs functionality to an DBMS implies that the system can be scaled to larger dimensions, both when it comes to query processing as well as to storage capacity. An extensible DBMSs should ease this integration by allowing for he necessarily extensions, identified below.

6.4.1 SDDS requirements on a DBMS

A transparent and efficient integration of SDDSs into a DBMS, without complete redevelopment of the DBMS kernel, poses several requirements on its functionality. The DBMS must:

\(^1\)http://www.cwi.nl/~monet
• be extensible at multiple levels, i.e. enable addition of new data types, algorithms and operators. Unfortunately, few (commercial) DBMS provide sufficient functionality in this area so far.

• provide a general communication package, such that the SDDS implementation does not have to “know” about what data is transported. Such a module can itself be an extension.

• transparent handling of native and user defined types, for example to be able to use native tables as building blocks for the management of the data held within the SDDS.

Ideally, the SDDS can be used in very much the same way as non-distributed data. In a database perspective it means that SDDSs should exhibit the same behavior as operations on its native tables. Further on, table operators has been extended with their semantically equivalent counterparts that hide the effects of the SDDS data distribution. SDDSs just become another data type, with the normal relational operators defined upon it.

6.4.2 Resource Management

We assume that each SDDS has a logical numbering of its participating nodes numbered 0..n − 1 by its algorithm or a mapping thereof, n being the number of nodes employed. When a node is addressed, we map the logical number to a virtual machine number. This virtual number is then translated to the unique machine number, which in turn can be used to get to the internet address (IP-number). This is a convenient way to cluster different SDDS’s data so that they use the same physical distribution, “syncing” on the primary key.

For example, if two relations (SDDSs) are indexed on the same key, then it makes sense to cluster data in such a way that all tuples with the same keys are kept at the same physical node. Also, SDDS load balancing [WBW94] can benefit from a strategy where several logical nodes of the same SDDS are mapped onto the same machine. This information is kept between different loads of the database. Therefore the same clustering and load balancing effect can be achieved at the next load onto a different set of physical machines.

6.4.3 SDDS Administration

The management of the SDDS requires some administrative information. It is used to access the data stored in the structure. Below we list the information as it was needed in Monet.
the *home location*, the virtual machine number where the zeroth (0th) logical server node of the SDDS is kept,

- the *mapping* from logical nodes to virtual machine numbers,

- the *unique identity number* of the SDDS,

- the clients current *image* of the SDDS,

- and finally, servers store the *distributed data* of the SDDS.

We introduce two user defined types into Monet to support SDDS. First, a client data structure, contains the necessary state information. It also acts as a handle to a binary relation containing the mapping from logical node to virtual machine. Second, a server data structure that contains the identity of the distributed table, the logical number of the server, and a handle to the table containing the data stored by that node of the SDDS. Details can be found in [KK97].

### 6.4.4 Algebraic Operations

Studies on SDDSs have focussed on individual tuple access. In [SAS95] they investigate the usage of an SDDS for a hash join algorithm, PJLH. Their algorithm overcomes previous drawbacks of static hash joins by adopting the number of participating nodes to the workload. The results are then extended for multi-joins. The join sites of PJLH are disjoint from the sites storing the participating join relations. However, in our setting, this is insufficient. Instead, we have added extended relational operators to deal with the SDDSs storage layout. For our implementation we merely assume that a number of relations has been chosen to be distributed using a SDDS schema and, in many cases, the structure is inherited by the result. Querying these relations is transparent.

The operators have been chosen for ease of implementation, because we are primarily interested in the overhead incurred by the use of SDDSs on the database kernel. In the experiment we focus on the *select* and *join* operators. They are the key operators needed to support SQL-like query languages. Furthermore, their implementations show characteristics typical for a large group of relational operations. An informal semantic description is shown below, where $A, B$ are binary relations and $a, b, c, d$ are (scalar) data, such as integers or strings:

<table>
<thead>
<tr>
<th>operator</th>
<th>definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>select($A$, $h$, $l$)</td>
<td>${(a, b) \in A</td>
</tr>
<tr>
<td>join($A$, $B$)</td>
<td>${(c, d)</td>
</tr>
</tbody>
</table>
6.5. IMPLEMENTATION AND PERFORMANCE STUDY

The select operator is rather straightforward to implement on an SDDS. Data is sent when needed to SDDS nodes for processing together with the SDDS data. The processing takes place in parallel, and results are then sent back to the originator of the operation. join operators, however, require more implementation effort, especially in the case of join over two SDDS-based tables. The different settings and ideas used are elaborated.

In the examples below all SDDSs have been distributed on the key attribute, i.e. SDDS = \{(a, b)\}, where the distribution is based on the key attribute \(a\).

- \textbf{select(SDDS, low, high)} \rightarrow \textbf{table}
  
The select operator \textit{broadcasts}\(^2\) a local select to all sites of the SDDS. Results are returned to the node that initiated the query.

- \textbf{join(SDDS, table)} \rightarrow \textbf{table}
  
The table is broadcasted to all nodes of SDDS where a local join takes place. The result is then returned to the originating node of the query.

- \textbf{join(table, SDDS)} \rightarrow \textbf{table}
  
The table is distributed on its join-attribute onto the same number of nodes of the SDDS. Effectively this turns into a hash-join. Results are then returned.

In the algorithms above, a substantial improvement could be achieved for certain further operators if the results were kept distributed. The decision to keep the results distributed or not depends on a number of parameters, and requires a cost based model, in general.

6.5 Implementation and Performance Study

The implementation is such that other SDDSs can easily be included. Since the dominant cost is network transport we expect similar performance characteristics, however. We now report on results obtained by the integration of LH* in Monet.

The experimentation is geared towards uncovering implementation problems and to obtain a first assessment of the overhead incurred in distributed processing under the SDDS. We want to show the following:

- Optimal size of a distributed partition,

\(^2\)Broadcasting to "all" nodes of an SDDS requires special care. The client does not normally know all participating nodes. We use the inherit structure of the splitting pattern to forward broadcasts, with addition of extra messages that incrementally update the client. In the end, leaving the client with enough information to complete the operations.
• Overhead added by SDDSs,
• Performance scalability.

We use a network multicomputer [Cul94] [Tan95] in our case a number of Silicon Graphics O2s, running IRIX6.3, each having at least 64 MBytes of memory. For communication the office network is used. This network is a mix of ATM-switches and Ethernet. Each workstation has a 180 MHz, R5000 MIPS CPU. Measures are given in milliseconds (ms). The experiments are run a number of times, this to decrease the disturbance from other uses of the network and computers, the best results are then kept and used for the graphs.

Loading of a database may be done in several different ways. If there is only one source, it could be segmented and the data could be loaded N-way parallel. We will not go into further details of different ways of loading data distributedly. We assume that when the queries are run that appropriate starting relations have already been loaded and distributed.

We use two tables big and t5. big is an SDDS table stored over a number of nodes, whereas t5 is a main memory table at the front node. The size of the table t5 is fixed to 100 000 entries (800 KBytes). The contents of both tables are pairs of integers (int, int). Values are unique, and data is not stored sorted. During the query processing indices may be created when it benefits operators.

The select operator scans the whole SDDS (LH*-)-table to find the matching values. In all our experiments we assume only the SDDS to be distributed. Operators that operate on local data and SDDS data, for example a join between a local table and an SDDS table, first get the data sent to the relevant SDDS nodes. These nodes execute the operator in parallel, sending back the results, to be combined. Note, that the timing in our experiments include the time for collecting the result at a single node.

### 6.5.1 Optimal Size of a Distributed Partition

A large file, larger than main memory, cannot be searched with high performance, if it fully resides on a single node. We investigate for a number of operators the behavior for increasingly larger datasets to find the point where their performance degrades into that of a disk-based system. This gives us the maximal size of a partition for a distributed table, under both network access and CPU cost.

In Figure 6.1, we show the time to execute range selection and joins using an increasingly larger relation. The range selection selects values in the interval 1 to 10 on the SDDS, and the two different joins. On the x-axis the file size is shown in MBytes, and on the y-axis the time in ms.

There is a big performance gap in scanning a 48 MBytes table compared to a 40 MBytes table. This illustrates that the table/file cannot be kept
wholly in main memory anymore.

Joins were made with table t5. The cost is approximately linear up to a file size of 48 MBytes. For larger tables the performance degrades quickly, because the datasets do not fit entirely into main memory. We studied the

<table>
<thead>
<tr>
<th>Bytes</th>
<th>#pagefaults</th>
<th>elapsed [ms]</th>
<th>user [ms]</th>
<th>system [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 MB</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>16 MB</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>24 MB</td>
<td>2</td>
<td>1344</td>
<td>1240</td>
<td>40</td>
</tr>
<tr>
<td>32 MB</td>
<td>5</td>
<td>1874</td>
<td>1660</td>
<td>80</td>
</tr>
<tr>
<td>40 MB</td>
<td>213</td>
<td>3060</td>
<td>2070</td>
<td>120</td>
</tr>
<tr>
<td>48 MB</td>
<td>11408</td>
<td>66 000</td>
<td>2490</td>
<td>1900</td>
</tr>
<tr>
<td>56 MB</td>
<td>13676</td>
<td>122 000</td>
<td>2930</td>
<td>3060</td>
</tr>
<tr>
<td>64 MB</td>
<td>16037</td>
<td>103 000</td>
<td>3320</td>
<td>2920</td>
</tr>
</tbody>
</table>

Table 6.1: Memory page faults, and elapsed/user/system time for the operation.

number of memory faults — number of pages need to be swapped in from disk — to more clearly understand the actual performance degradation. As shown in table 6.1, for the select(1,10) operation the time increases slowly for smaller sets of data up to 40 MBytes of data. At 48 MBytes and beyond the number of pagefaults clearly corresponds to the size of the data set, which means that all of the data have been swapped in from the disk. The table also shows the elapsed time, user CPU time, and system CPU time. The latter two grow linearly with the increase of data, whereas the elapsed time indicates waiting for disk. Our solution, which is presented in the following text, is to distribute the data using an SDDS, LH*, and we
show we can query even larger sets of distributed data within the same time.

### 6.5.2 Overhead added by SDDSs

In this experiment we keep the current file size below 40 MB to ensure memory residence of the data. Data is stored on one node and queried remotely. Interestingly, the overhead for scanning `select(1,10)` is kept reasonably low in the experiments conducted, as shown in Figure 6.2. The overhead of a scan (`select`) operation on the file stored at a different node is limited to around 300 ms for the different sized files (8 MB, 16 MB, 32 MB). This includes the time to send back the results. Joins, however, lose in performance directly, because the amount of data to be transferred to the remote node.

![Figure 6.2: One node, distributed access, varying sizes.](image)

### 6.5.3 Performance Scalability

We show the performance by varying two parameters.

- The size (cardinality) of the SDDSs stored.
- The number of server nodes (workstations).

In the first experiment, we keep the size constant at 1M entries giving a table using 8MBytes. The number of nodes involved is varied from 1 to 16, consequently.

The experiment confirms that employing more nodes to store the same amount of data is beneficial for scanning, see Figure 6.3. Range value scanning even improves in performance linear with the number of nodes. Noticeable is that the time directly halves for one of the joins `join(big, t5)`
using one more node. Then the cost increases somewhat for both of the joins due to sending t5 to more nodes. However, this cost does not increase sharply, which means that more data can be stored at more nodes without too much overhead in the cost. This holds under the assumption that the temporary data and the partition of the distributed relation fits into the memory available.

The second experiment varies the size from 1M entries to 8M entries, with a database size of 8MBytes to 64MBytes. The number of nodes is kept constantly at 8. Figure 6.4 illustrates that an increase of data on a fixed number of nodes leads to a modest increase of the querying cost. Scanning is very fast, much faster than using local main memory of a single node,
since it is executed in parallel over the SDDSs nodes. Join cost increases slowly, linear to the size of the transported data.

The last experiment keeps the ratio of entries and number of nodes constant. 4M entries are stored at each node. The file size varies from 4M entries to 32 M entries, giving 32MBytes to 256MBytes and 1 to 8 nodes.

![Figure 6.5: Scale up values; Varying number of nodes, each storing 32 MB.](image)

From Figure 6.5 we see that keeping the same amount of data on all nodes, keeps the querying cost vaguely constant. The higher costs comes from the involvement of more nodes. Part of the time increase for joins is explained by the cost of distributing the t5 table to a larger number of nodes. Observe that for $\text{join(big, t5)}$ we can query nearly 3 times as much distributed data in about the same time compared to querying main-memory data, this is shown in Figure 6.6. $\text{Select(5)}$ shows a slight increase from 174 ms, 1 node, to 273 ms for 8 nodes. Surprisingly, the time to execute $\text{select(1,10)}$ is extra ordinarily constant\(^3\), around 2000 ms. However, an reasonable stable time is not unlikely in view of the declining numbers in the constant sized experiments (Figure 6.3), and the increasing numbers in the constant nodes experiments (Figure 6.4).

### 6.5.4 Discussion

The overall conclusions from the experiments are shortly:

- A partition storing distributed data using an SDDS in Monet should not exceed approximately 40 MB on our 64 MB machine, i.e. the memory free for user processes. This keeps the performance from degrading from main memory to disk based with trashing.

\(^3\)There was one machine that dominated in response time, that gave the same best time for different sizes of the table.
6.6. **SUMMARY**

![Figure 6.6: Comparison between main-memory processing, and distributed memory.](image)

- The SDDS related overhead added by our integration in a fully fledged DBMS kernel is low. Scanning when employing an increasing number of nodes excel over the non-distributed case with perfect scalability.

- For a fixed sized file, joining shows a moderate increase in the cost when a larger number of nodes is used.

- When using a variety of different file sizes, for a fixed number of nodes, the costs are higher for larger files. However, it increases much slower than linearly. For example, comparing joins on 1 M entries (8MBytes) and joins on 4 M entries (32 MBytes), the cost increase with only 33% to 46%, for our different joins.

- A file can easily be scaled. I.e, a larger number of nodes is used for a larger amount of data, keeping a constant load on each node. Querying of data is done vaguely in constant time, independent on the amount and the number of nodes.

### 6.6 Summary

The prime novelties we have shown in this chapter is — by analysis and implementation — that Scalable Distributed Data Structures provide a viable alternative to conventional data distribution schemes, based on static hash- and range-fragmentation. LH*, a well-known SDDS, has been integrated with an extensible database system — Monet. The key relational operators were made SDDS aware, such that the query optimizer is relieved from the
expensive task to a priori select the 'best' data fragmentation and distribution scheme.

We identified the core requirements on an extensible database system, in order for an SDDS to be integrated. In our implementation platform, Monet, the integration was facilitated by the already-present standard modular extensibility. The extension module added relational operators, that are able to cope with the distributed SDDS-based tables. In the end, it means that distributed data storage can be treated without any textual/syntactical changes to the queries compared to native (local) data.

The performance experiments demonstrate that the overhead incurred by the SDDS itself is minimal. The bulk processing cost stems from moving large fragments of data around. However, in most realistic cases of distribution, querying distributed memory is faster than accessing disk-based data. This makes it possible to use workstation clusters to provide for database storage.

This study is currently expanded towards a run-time optimization technique based on a cost-model, and plans to perform further experiments using queries from TPC-D on an SP/2 platform using our SDDS storage.