Scalable Storage for a DBMS using Transparent Distribution

J.S. Karlsson, M.L. Kersten

Information Systems (INS)

INS-R9710 December 31, 1997

Report INS-R9710
ISSN 1386-3681

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

Scalable Storage for a DBMS using Transparent Distribution

Jonas S. Karlsson, Martin L. Kersten
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
jonas@cwi.nl, mk@cwi.nl

ABSTRACT

Scalable Distributed Data Structures (SDDSs) provide a self-managing and self-organizing data storage of potentially unbounded size. This stands in contrast to common distribution schemas deployed in conventional distributed DBMS. SDDSs, however, have mostly been used in synthetic scenarios to investigate their properties.

In this paper we concentrate on the integration of the LH* SDDS into our efficient and extensible DBMS, called Monet[1]. We show that this merge permits processing very large sets of distributed data.

In our implementation we extended the relational algebra interpreter in such a way that access to data, whether it is distributed or locally stored, is transparent to the user. The on-the-fly optimization of operations — heavily used in Monet — to deploy different strategies and scenarios inside the primary operators associated with an SDDS adds self-adaptiveness to the query system; it dynamically adopts itself to unforeseen situations. We illustrate the performance efficiency by experiments on a network of workstations. The transparent integration of SDDSs opens new perspectives for very large self-managing database systems.

1991 Computing Reviews Classification System: H.2.4 : Systems – Query Processing, D.4.3 : File Systems Management – Distributed File Systems, C.2.4 : Distributed Systems – Distributed Databases
Keywords and Phrases: DBMS, Scalable Distributed Data Structures, Distributed/Parallel Databases, Distributed Query Processing, Client/Server Architecture
Note: Partially funded by the HPCN/IMPACT project.

1. 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 distribution of data and optimizers to cope with it. Despite this, modern applications, such as GIS and Data Mining, continue to stress the need for better techniques; both in terms of scalability and performance. Especially the limited and rigid scheme deployed for distributed and parallel data handlers and resulting complexity of the query optimizers, hinder major breakthroughs.

Scalable Distributed Data Structures (SDDSs) [LNS93] [LNS96] 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 nodes. Most studies have been focused on algorithm and experiments in an isolated context, i.e., not integrated in a database system.

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 SDDS functionality can be exploited in the algebra to choose from a set of strategies for excelling in performance. The strategy chosen depends on several dynamic properties, such as locality of data, their size, and costs of partitioning and distributing.

Their integration solves a problem often encountered in distributed systems, where manual (re-)configuration is required to handle larger and larger data sets. SDDS storage scale automatically

[1] See http://www.cwi.nl/~monet

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 2 we shortly describe the key properties of Scalable Distributed Data Structures and the Monet database system. Then, in Section 3, we describe the implementation and its integration of SDDSs in Monet. The influence of Monets' algebraic components and dynamic operator optimization scheme is analysed in Section 4. Section 5 shows preliminary performance measures of our implementation. Finally, in Section 6 we conclude this work and give directions for future research.

2. Background

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

2.1 SDDSs

Scalable Distributed Data Structures (SDDSs) [LNS93] can be classified as an general access path mechanism. Several SDDSs has been defined [Dev93] [WBW94] [KW94] [LNS94] [KLR96]. 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 requesting the tuple and one message for sending the data.

SDDSs differs from other distributed schemas in that it allows the number of nodes to increase with a very small cost. Many other distributed data structures requires a costly total re-organization for adding a node because they employ static distribution schemas, examples are round-robin [Cor88], hash-declustering [KTMO84], range-partititioning [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 they 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 continously, 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 conclusion the main features of an SDDS are:

- There is no central directory that clients have to go through for data access. This avoids hot-spots.
- 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.

The operations directly supported are limited to individual tuple access. It requires generalization to support a relational algebra, which is addressed in this paper.

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.

2.2 Monet

Monet [BK95] 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.
- Employs main-memory algorithms for querying/processing.
- Extensive use of OS virtual memory primitives to manage larger sets.
- Extensibility through modules providing support for domain specific data structures, indices, and operators.
- Relational algebra operators implementations with dynamic run-time optimizations — "Live Optimization".
- High throughput achieved using bulk operator processing. This enables Monet to benefit most from modern computers cache-line behavior, on the expense of that intermediate results are fully materialized.

Monet is heavily used in Data Mining applications[HKM95] and GIS[BQK96], and its supreme performance has been demonstrated against several benchmarks, including OO7[BKK96] and TPC-D[BWK98]. The current implementation runs on workstations and exploits parallelism of SMP machines. However, shared memory computers provide limited means to scale. Therefore, several activities are underway to exploit MPP (Massivly Parallel Processing) machines to deal with TeraByte problems. In this paper we focus on the SDDS approach to tackle the distribution issue.

Monet features of direct relevance to this paper are the concepts of ATOMs and BATs. ATOMs are user definable abstract data types, i.e. non-modifiable data, such as strings, integers, and floats. The ATOMs are stored in tables. Since Monet employs a fully decomposed storage schema, tables only contains <`key,value`> pairs. Such a table is called a *Binary Association Table*, BAT for short. The BATs are operated upon using an extended relational algebra.

The system is heavily optimized for these algebraic operations. In particular, the operators decide on-the-fly on the best algorithm to perform their task, including construction of temporary search accelerators, OS memory advice, etc..

3. SDDS within Monet

Adding SDDSs functionality to Monet implies that the system can be scaled to larger dimensions, both when it comes to query processing as well as to storage capacity. This can be achieved mostly with the extensibility already provided by Monet system. We show how these *already-in-the-box* features allows us to integrate LH* into Monet. In particular, we address issues related to distributed processing and management of the SDDS dictionary information.

3.1 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 should 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.
- The DBMS should provide a general communication package, such that the SDDS implementation does not have to "know" about what data is transported. This is supported in our system.
- The DBMS should support storage of any kind of data, i.e., to use native tables as building blocks for our management of the data held within the SDDS.

To fully benefit from an SDDS, the database system should In addition provide a means to adjust the query execution plan. In particular, it should provide accurate optimization information to select the algorithm most fit for a given operation. For this we identify two strategies: a *traditional* approach and *Live Optimization*.

The *traditional* approach is to extend the DBMS's query optimizer with knowledge (cost modelling) that distributed relations are potentially more expensive to use, and then do parallel query optimization. It would have to make plans how to place, gather, allocate data and resources in an optimal way. A drawback with such a (static) plan is that it usually does not adapt itself gracefully to new circumstances. For example, errors in estimates of cardinality of multi-joins are easily orders of magnitudes off[IC91].

Live Optimization, on the other hand, liberates the optimizer from the burden of physical aspects. It provides a coherent interface to bulk operators on data, whether distributed or not, and it hides optimization decisions in the operators themselves; they do their best in being efficient. This is achieved by allowing operators to select from different algorithms to produce the result. Examples of this are creating or using indices when applicable, sorting data to allow for merge-joins, etc.. This kind of dynamic optimization is already successfully used in Monet for non-distributed operators.

3.2 SDDSs in Monet

There are many aspects of storing and using SDDSs in DBMSs. Ideally they can be used in very much the same way as non-distributed data. In a Monet perspective it means that SDDSs should exhibit the same behavior as operations on Binary Association Tables (BATs).

The transparency, in the Monet case, is partially achieved by its automatic builtin type dispatching[2]. The target language to which queries are compiled is built on a basis that the interpreter can resolve overloaded functions. Therefore, the obvious strategy is to make an LH* ATOM or an abstract data type in Monet. This type then has to implement all (relevant) methods defined over BATs. But, whereas operations on relations occur in main memory, the operations on SDDSs have to be distributed to where the data is stored. We will sketch the implementation of the most interesting algebraic operators in Section 4.

We use the general mechanism provided by the Monet interface to "override" algebraic operators with more subtle algorithms. The algorithm for resolving overloaded functions will select the appropriate one, especially crafted for SDDSs. All relevant operators on relations are extended with their counterparts that hide the effects of the distribution given by the SDDS. SDDSs just become another type, with the normal relational operators defined upon it.

3.3 Resource Management

Each SDDS has a logical numbering of its participating nodes usually numbered $[0..n-1]$ by its algorithm, n being the number of nodes employed. When a node is addressed, the logical number is mapped to a *virtual machine number*. This virtual number can be translated to the unique machine number, which in turn can be used to get to the internet machine address. This is a convenient way

[2] Monet allows addition of user defined "commands". The interpreter selects commands based on the types of the arguments, whereas for scripting procedures it does not.

to cluster different SDDS's data so that they use the same physical distribution, "syncing"[3] on the primary key.

For example if two relations (SDDSs) are indexed on the same OID. It makes sense to always cluster data in such a way that all tuples with the same OID are kept at the same physical node. By letting the SDDSs have the same mapping, and only changing the mapping of virtual machine number to physical node, this clustering effect is achieved. Also, SDDS load balancing [WBW94] can benefit from a strategy where severa logical nodes of the same SDDS are mapped onto the same machine. This information is part of the SDDS dictionary, and is therefore kept between different loads of the database. No physical node information needs to be updated to achieve the same cluster/load balancing effect at the next load onto, possible, a different set of physical machines.

For allocation of machines to the SDDSs we currently use a central node. It keeps track of all SDDS nodes and establishes a mapping from physical identity (such as an Internet address) to a *unique machine number*.

3.4 Storage Demands for the SDDS itself

The SDDS dictionary; the information used to access data stored contains the following:

- the *home location*, the virtual machine number where the zero'th 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.

On the client-side we represent all data needed using a *handle*. The handle is an ATOM of type `lhslh`. Methods on the type `lhslh` implement algebraic operations. The actual data is stored distributed on several *server*-nodes. A server needs access to the same information about the SDDS as a client, apart from that it stores the actual data. We therefore let the server-handle, `lhslh_server`, give access to the client-handle. At each physical site we store only one client information for each SDDS. Server nodes, however, may store several logical servers nodes of an SDDS, all of which will share the same client information.

In Monet, complex objects are usually stored vertically decomposed into a number of tables, i.e. BATs. The SDDS dictionary is a complex object and, therefore, could be decomposed. However, we do not see SDDSs dictionaries themselves as interesting objects for querying, so we choose to implement SDDS handles using ATOMs. This proves to be an interesting decision. When the image of a client is updated we have to replace the ATOM[4]. For Monet this will actually show up as an advantage rather as an disadvantage, even though that it seems a bit cumbersome at first.

In a multi-processing environment such as Monet, there is an advantage in that ATOMs cannot be updated, it means that the SDDS information does not need to be locked. For example, if multiple threads look up the same ATOM they will be provided with their own copy. Since all threads have their own copies of the same data, no locks has to be applied. This simplifies implementation. At the downside, if a thread retains the value too long its actions might be somewhat outdated. The thread will eventually have to update it at convenient points. However, SDDS implementation schemes ensure that a client's request will reach its final correct destination, possibly with some additional overhead.

To conclude, we introduced two new atoms into Monet to support SDDS. `lhslh`, the client atom, contains the necessary state information. It also acts as a handle to a BAT containing the logical node

[3] In a fully decomposed storage schema, full clustering of associated data on the same physical nodes would, for example, require synchronous splitting of servers. Still, even if this is not done, many operators will be more efficient.

[4] Remember, we cannot update it, but we can update the table where it is stored.

to virtual machine mapping. The second atom, `lhslh_server`, contains the identity of the distributed relation, the logical number of the server, and a handle to the BAT containing the data stored by that node of the SDDS.

4. Algebraic Operations

Studies on SDDSs were mostly focussed on individual tuple access. PJLH [SAS95] investigates the usage of an SDDS for storing the output of a distributed (hash-) joins, 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 some 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 made transparent. The query optimizer merely generates code as was the table stored locally. Although this simplifies the optimizer, it does not necessarily lead to the most optimal execution plans. However, this choice aligns with the research hypothesis of the Monet group — to relieve the optimizer as much as possible by extensive dynamic support within the operator implementation.

For example, the Monet `select` operator performs many runtime optimization decisions, such as, creating indices when doing lookups, sorting when it is considered to be beneficial, etc...

For this experiment we focus on the `select` and (`semi-`) `join` operators. They are the key operators needed to support SQL-like query languages. Note that the binary representation of the tables stresses the need for semi-joins instead of the traditional projection operator. Futhermore, their implementations show characteristics typical for a large group of relational operations. An informal semantic description is shown below (A, B are BATs and a, b, c, d are ATOM)view:

operator	"definition"
select(A, h, l)	$\{(a,b) \in A \mid l \leq b \leq h\}$
join(A, B)	$\{(c,d) \mid (a,b) \in A, (c,d) \in B, b = c\}$
semijoin(A, B)	$\{(a,b) \mid (a,b) \in A, (c,d) \in B, a = c\}$

The `select` operator is rather straightforward to implement on an SDDS. The parameters are sent to all nodes storing buckets of the SDDS, where the data is scanned, results are either stored locally or sent back to the originator of the operation. In a `semijoin` one or both of the relations A and B may be distributed but it is rather efficient and simple to implement. However, the `join` operator requires more implementation effort, especially in the case of join over two SDDS-based tables. In the next section we outline the different settings and ideas used.

4.1 Live Optimization

The SDDS implementation generates a large number of options for *Live Optimization*. It includes runtime decisions on materializing of distributed data onto one node, or redistribution of data according to another schema, creating indices when operators do benefit, etc..

The granularity of execution in Monet is at the level of algebraic bulk operators. Operators are executed in sequence. Intermediate results are stored fully materialized. Live optimization allows us to use the same plans generated for a non-distributed environment without having to do any rewriting, or further restructuring. These keeps flexibility in the plans by deferring decisions until more information is known. For example, for performance reasons there may be a great difference if intermediate results are stored distributed or not. By default, results are stored distributed. The result can later be redistributed or collected onto one node if succeeding operators find it beneficial.

The current implementation uses the following strategies. They have been chosen for ease of implementation, because we are primarily interested in the overhead incured by the use of SDDSs on the database kernel.

- `select(SDDS1, low, high)` → $\texttt{SDDS}_2$

  The `select` operator broadcast a normal select to all sites of the $\texttt{SDDS}_1$. Results are stored where generated and a handle, $\texttt{SDDS}_2$, is returned. The generated SDDS is a subset of $\texttt{SDDS}_1$, but with the same distribution.

- `join(SDDS1, BAT)` → $\texttt{SDDS}_2$

  The `BAT` is broadcasted to all nodes of $\texttt{SDDS}_1$ where a local join then takes places. The result is kept on the same node as it was generated, i.e., the $\texttt{SDDS}_2$ is distributed in the same way as $\texttt{SDDS}_1$. Broadcasting the `BAT`, is practically costly. If the nodes of the $\texttt{SDDS}_1$ store less data, the data can be materialized on the node where the `BAT` resides, giving a local join and local result.

- `join(BAT, SDDS)` → `BAT*`

  The `BAT` is distributed on its join-attribute onto the same number of nodes of the SDDS. Effectively this turns into a hash-join. Results are stored distributed, and are "randomly" organized. Again, if the `SDDS` is "small" an alternative is to materialize it as an BAT, and join locally.

- `join(SDDS1, SDDS2)` → $\texttt{SDDS}_3$ `or BAT*`

  The first alternative is to hash-join the $\texttt{SDDS}_1$ data onto the nodes storing $\texttt{SDDS}_2$. The result is "randomly" distributed on the nodes of $\texttt{SDDS}_2$.

  The second alternative is to broadcast $\texttt{SDDS}_2$ to all nodes of $\texttt{SDDS}_1$. The result $\texttt{SDDS}_3$ is then distributed similarly as $\texttt{SDDS}_1$ on the same nodes.

  Again, depending on the sizes of the SDDSs, either of them could be materialized and then broadcasted to the other.

- `semijoin(SDDS1, BAT)` → $\texttt{SDDS}_2$

  This semijoin hash-joins the BAT onto the nodes of $\texttt{SDDS}_1$. The result $\texttt{SDDS}_2$ is a subset of $\texttt{SDDS}_1$, distributed in the same way on the same nodes.

- `semijoin(BAT, SDDS1)` → $\texttt{SDDS}_2$

  Again, we hash-join the `BAT` onto the nodes of $\texttt{SDDS}_1$. The result $\texttt{SDDS}_2$ is a subset stored similarly on the same nodes as $\texttt{SDDS}_1$.

- `semijoin(SDDS1, SDDS2)` → $\texttt{SDDS}_3$

  This semijoin is most efficient if the both SDDSs use the same mapping (and same number of servers). Then we semijoin locally. The result will already "appear" distributed correctly.

  If the SDDSs resides on a disjoin set of servers using different mappings we send the SDDS that incurs the least communication onto the other SDDS. The result will again "appear" distributed.

As can be seen from above, in many cases the results "appear" distributed, and can hence be used as SDDSs. However, their load may indicate that they either should be shrinked (employ less nodes) or be expanded.

5. Implementation and Performance Study

In this section we report on preliminary results obtained by 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 with live optimization. We want to show the following:

- Optimal Size of a Distributed Partition
- Overhead Added by SDDSs

- Performance Scalability

We use a network multicomputer [Cul94] [Tan95] — in our case a number of Silicon Graphics O_2s, running IRIX6.3, each having 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. All measures are given in milli seconds (ms).

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, and we assume that when the queries are run that appropriate starting relations have already been loaded/distributed, and thus are main memory resident.

We make use of two tables `big` and `t5`. `big` is an SDDS table stored over a number of nodes, wheras `t5` is a main memory table at the front node. The size of the table `t5` is fixed to 100 000 entries. 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 operators find it beneficial.

5.1 Optimal Size of an 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 some operators the behaviour for larger and larger datasets to find the breakpoint where their performance degrades into that of a disk-based system. This gives us the optimal size of a partition for a distributed BAT is.

#	Bytes	.select(5)	.select(1,10)	join(t5, big)	join(big, t5)
1 M	8 MB	276	427	498	2846
2 M	16 MB	553	896	516	5653
3 M	24 MB	866	1302	498	8804
4 M	32 MB	1160	1763	774	12042
5 M	40 MB	1582	2216	965	17846
6 M	48 MB	15777	14874	1203	22994
7 M	56 MB	17885	18043	7423	26864
8 M	64 MB	19932	19686	6073	33892
9 M	72 MB	22017	21489	10126	36898
10 M	80 MB	33340	24174	5585	37829
11 M	88 MB	28474	26714	13993	43783

The first column is the number of element stored in the LH* file, the second the file size in Bytes, third column is the cost of a `select` of an fixed value, fourth column is the cost of selection of an interval. The operator `select` uses scanning of the whole LH*-table to find the matching values. And the last two columns contains two different joins. `t5` is a BAT with integers, containing approximately 100 000 entries (800 KBytes), and `big` is the distributed LH* file. In all our experiements we assume only the SDDS to be distributed. All other data is broadcasted to all nodes, which execute the operators in parallel, sending back the results. Then the results are combined. All timing vaules are shown in milli seconds (ms). Note that the timing in our experiments *include* the time for collecting the result. Thus the results are not kept distributed.

As can be seen on scanning of a 48 MBytes table compared to a 40 MBytes table, there is a big gap in performance. This illustrates that the table/file cannot be kept wholly in main memory. It is an result from the limited main-memory, 64 MBytes, in our workstations.

Joins were done on a table, `t5`, with 100 000 entries (800KBytes). The cost of the join approximately increase linear upto a file of 48 MBytes. From 48 MBytes and upwards the cost is higher. These datasets do not fit entirely into the main memory. Thus the performance degrades.

#	Bytes	pagefaults	elapsed	user	system
1 M	8 MB	1	-	-	-
2 M	16 MB	2	-	-	-
3 M	24 MB	2	1344	1240	40
4 M	32 MB	5	1874	1660	80
5 M	40 MB	213	3060	2070	120
6 M	48 MB	11408	66 000	2490	1900
7 M	56 MB	13676	122 000	2930	3060
8 M	64 MB	16037	103 000	3320	2920

To more clearly understand the actual performance degradation that occurs at 48 MByte usage, we studied the number of memory faults (number of pages needed to be swapped in). We studied it for the `select(1, 10)` command. As can be seen above, it increase slowly for smaller sets of data at 5 M entries we see a small rise, and at 6 M entries (48 MBytes) and beyond the number of pagefaults clearly corresponds to the size of the data set. The table also shows a case where we measured the elapsed time, user CPU time, and system CPU time. The latter two grows linearly with the increase of data, whereas the elapsed time indicates disc input waiting. Times are shown in ms.

5.2 Overhead added by SDDSs

The overhead imposed in using SDDSs, is measured by taking files of the same sizes as above, but distributed to *one remote* node.

#	Bytes	.select(5)	.select(1,10)	join(t5, big)	join(big, t5)
1 M	8 M	198	781	8378	5191
2 M	16 M	200	1200	11952	7939
4 M	32 M	174	1999	20553	13427
8 M	64 M	112	107781	crash	crash

Interestingly, the overhead for scanning, `select(1,10)`, is keept reasonably low in the experiments. The overhead for a file of 40 MByte is only approximately 380 ms. Joins, however loose in performance directly, since the amount of data needed to be transferred, table `t5`, to the added nodes, is much larger. For 64 MB joins, the memory was exhausted so no figures are availiable.

5.3 Performance Scalability

We show the performance by varying two parameters.

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

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

#nodes	#	.select(5)	.select(1,10)	join(t5, big)	join(big, t5)
1	1 M	198	781	5191	8378
2	1 M	198	388	5430	4026
4	1 M	200	400	6851	6737
8	1 M	200	330	9315	8955
16	1 M	320	1363	22394	21492

This experiment confirms that employing more nodes for storing the same amount of data is in some cases beneficial. This is true for scanning. For joining the increased cost stems from sending `t5` to more nodes. However, the cost does not increase linearly which means that more data can be stored at each node without too large an overhead in the cost.

The second experiment varies the size from 1M entries to 32M entries (8MBytes to 256MBytes). The number of nodes is kept constantly at 8.

#nodes	#	.select(5)	.select(1,10)	join(t5, big)	join(big, t5)
8	1 M	200	330	9315	8955
8	2 M	237	399	9909	9858
8	4 M	248	398	11196	11219
8	8 M	218	618	13597	11956
8	32 M	273	1999	53145	42992

An increase of data on a fixed number of nodes modestly increases the querying cost. Scanning is very fast, much faster than using local main memory, since it is execute in parallel over the nodes. Join cost increases slowly, apart from the drastic figure for 32M entries.

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.

#nodes	#	.select(5)	.select(1,10)	join(t5, big)	join(big, t5)
1	4 M	174	1999	20553	13427
2	8 M	190	1999	22393	15721
4	16 M	227	1999	25414	20079
8	32 M	273	1999	53145	42992

Keeping the same amount of data on all nodes, keeps the querying cost vaguely constant. Part of the increase for joins is explained by the cost of distributing the `t5` table to a larger number of nodes. Again, at 32M entries the increase is higher than expected, and requires more investigation. Surprisingly, the time to execute `select(1,10)` is extra ordinarly constant. However, this is not unlikely in view of the declining numbers in the constant sized experiments, and the increasing numbers in the constant nodes experiments.

5.4 Conclusions

The overall conclusions from the experiments are shortly:

- A partition storing distributed data from an SDDS using Monet should not exceed approximately 40 MB on a 64 MB machine. This keeps the performance from degrading from main memory to disk based with trashing.
- The SDDS related overhead added by our integration is low. Scanning when employing an increasing number of nodes excell over the non-distributed case. Showing 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 and joins on 8 M entries, the cost increase with only 46 %.
- 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. Summary

The prime novelties of this paper is that it shows — by analysis and implementation — that Scalable Distributed Data Structures provides a viable alternative to conventional data distribution schemes.

LH*, a well-known SDDS, is integrated with the Monet database system. The key relational operators have been 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.

The integration was facilitated by the ease of extensibility of the Monet system. The module mechanism enables concise description of the SDDS dictionary information, and its relational interpreter is able to cope with overloaded functions to facilitate queries over SDDS-based tables. In the end, it meant that data being stored could be treated without any textual/syntactical changes needed over ordinary tables.

We have also indicated that *Live Optmization* — as pervasive in Monet — is a promising concept. Also for hiding and making operations on distributed data transparent. 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, distributed memory is faster than accessing disk-based data.

This study is currently extended to improve the runtime optimizations further and to compare our results on TPC-D with those obtained on an SP/2 platform using SDDS storage.

References

[BK95] Peter A. Boncz and Martin L. Kersten. Monet: An Impressionist Sketch Of An Advanced Database System. In *Basque International Workshop on Information Technology: Data Management Systems*, San Sebastian (Spain), July 1995. IEEE.

[BKK96] Peter A. Boncz, F. Kwakkel, and Martin L. Kersten. High Performance Support for OO Traversals in Monet. In *British National Conference on Databases(BNCOD'96)*, 1996.

[BQK96] Peter A. Boncz, Wilko Quak, and Martin L. Kersten. Monet And Its Geographic Extensions: a Novel Approach to High Performance GIS Processing. In *Advances in Database Technology — EDBT'96*, pages 147–166, Avignon, France, March 1996. Springer.

[BWK98] Peter Boncz, Annita N. Wilschut, and Martin L. Kersten. Flattening an object algebra to provide performance. To appear at the 14th International Conference on Data Engineering, February 1998.

[Cor88] Teradata Corporation. DBC/1012 data base computer concepts and facilities. Technical Report Teradata Document C02-001-05, Teradata Corporation, 1988.

[Cul94] D. Culler. NOW: Towards Everyday Supercomputing on a Network of Workstations. Technical report, EECS Technical Reports UC Berkeley, 1994.

[Dev93] R. Devine. Design and implementation of DDH: A distributed dynamic hashing algorithm. In *Procedings of the 4th International Conference on Foundations of Data Organization and Algorithms (FODO)*, 1993.

[DGG+86] D. DeWitt, R. Gerber, G. Graefe, M. Heytens, K. Kumar, and M. Muralikrishna. GAMMA: A high performance dataflow database machine. In *Procedings of VLDB*, August 1986.

[HKM95] M. Holsheimer, M. L. Kersten, and M. L. Mannilla. A Perspective on Databases and Data Mining. Montreal, Canada, 1995.

[IC91] Yannis E. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. In *International Conference on Management of Data*. ACM-SIGMOD, June 1991.

[KLR96] Jonas S Karlsson, Witold Litwin, and Tore Risch. LH*LH: A Scalable High Performance Data Structure for Switched Multicomputers. In *Advances in Database Technology — EDBT'96*, pages 573–591, Avignon, France, March 1996. Springer.

[KTMO84] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. Architecture and performance of relational algebra machine GRACE. In *Procedings of the Intl. Conference on Parallel Processing*, Chicago, 1984.

[KW94] B. Kroll and P. Widmayer. Distributing a Search Tree Among a Growing Number of Processors. In *ACM-SIGMOD International Conference On Management of Data*, 1994.

[Lit80] W. Litwin. Linear Hashing: A new tool for file and table addressing. In *Procedings of VLDB*, Montreal, Canada, 1980.

[LNS93] W. Litwin, M-A Neimat, and D. Schneider. LH*: Linear hashing for distributed files. ACM SIGMOD International Conference on Management of Data, May 1993.

[LNS94] W. Litwin, M-A Neimat, and D. Schneider. RP*: A Family of Order Preserving Scalable Distributed Data Structures. VLDB Conference, 1994.

[LNS96] W. Litwin, M-A. Neimat, and D. Schneider. LH*: A Scalable Distributed Data Structure. *ACM-TODS Transactions on Database Systems*, Dec. 1996.

[SAS95] Vineet Singh, Minesh Amin, and Donovan Schneider. An Adaptive, Load Balancing Parallel Join Algorithm. Technical Report HPL-95-46, Hewlett-Packard Labs, 1995.

[Tan95] Andrew S. Tanenbaum. *Distributed Operating Systems*. 1995.

[WBW94] R. Wingralek, Y. Breitbart, and G. Weikum. Distributed file organisation with scalable cost/performance. In *Proc of ACM-SIGMOD*, May 1994.