AMOS-SDDS: A Scalable Distributed Data Manager for Windows Multicomputers

Yakham Ndiaye, Aly Wane Diène, Witold Litwin
CERIA Université Paris IX Dauphine, Place du Mal. de Lattre, 75775 Paris Cedex 16, France. Fax : +33 1 44 05 49 44 Phone: +33 1 44 05 40 47 {Yakham.Ndiaye, Ali-Wane.Diene, Witold.Litwin}@dauphine.fr

Tore Risch Uppsala Universitet, Dept. of Information Science, Box 513, Uppsala, SE-751 20, Sweden
Tore.Risch@dis.uu.se

Abstract

Known parallel DBMS offer at present only static partitioning schemes. Adding a storage node is then a cumbersome operation that typically requires the manual data redistribution. We present an architecture termed AMOS-SDDS for a share-nothing multicomputer. We have coupled a high-performance main-memory DBMS AMOS-II and a manager of Scalable Distributed Data Structures (SDDS) into a scalable distributed system. SDDS provides the scalable data partitioning in distributed RAM, supporting parallel scans with function shipping. AMOS-SDDS couples both systems using essentially the AMOS-II foreign function interface. The scalability that appeared from our experiments abolishes the cumbersome storage limits of a single site RAM DBMS technology. Its RAM query processing and scalable data partitioning are an improvement over the current parallel DBMSs technology. We validate AMOS-SDDS architecture by experiments with distributed nested loop join queries over a file scaling up to 300.000 tuples. It includes performance study of speed-up and scale-up characteristics. The results encourage the use of SDDS for modern high-performance database systems.

Key words: Multicomputer, scalability, distributed data structures, RAM database systems.

1 Introduction

Collections of popular PCs and WSs connected through typical 10/100 Mb/s networks became a standard. The concept of *(network) multicomputer* emerged to designate such configurations. A multicomputer offers potentially storage and processing capabilities rivaling a supercomputer at a fraction of the cost. Research on multicomputers became therefore popular.

Scalable Distributed Data Structures (SDDSs) are data structures specifically for multicomputers, [10]. They aim at new storage and processing capabilities. An SDDS is partitioned over some server nodes. Applications manipulate data from client nodes. The address computations do not involve any centralized directory. Data are typically stored in the distributed main memory (DRAM). An SDDS may easily handle many GByte files, accessible in a fraction of the disk access time. An SDDS scales to new sites through splits of those that fill up. Splits are transparently for the applications. All SDDSs support the key searches, some offer the range searches or multikey searches, and all provide also non-key parallel scans. For the latter capability, the client ships a function with the selection predicate or parameters. The servers return the selected records in parallel.

Several SDDSs are known. In particular, the LH* schemes provide the scalable distributed linear hash partitioning, [8], [10]. Likewise, the RP* schemes provide the scalable distributed range partitioning, [4], [11], [18]. Several prototypes have implemented selected SDDSs. The *SDDS* prototype that we design at CERIA is the most extensive such system, to the best of our knowledge. It runs on Wintel multicomputers and is designed for any SDDS. At present, it offers several variants of LH* and RP* schemes. Some are the high-availability schemes that tolerate multiple server failures [9].

High-performance database management uses basically two technologies. One is the RAM database, e.g., of the well-known Object-Relational DBMS *AMOS-II* [17]. A RAM database offers best access performance. It however of limited size and scalability, bound by a single node RAM capacity, usually two GBytes at present. AMOS II therefore is configurable as a distributed multi-database system where each node also may wrap external data sources [16]. In the experiments reported here we use only the single database AMOS-II configuration.

Another technology is the *parallel* database, typically on a share-nothing multicomputer or supercomputer (also often called now *switched* multicomputer). This technology allows for large sizes. It is however at present disk based; hence the database access is typically much slower than to a RAM database. The scalability is also limited. Known DBMS offer at present only static partitioning schemes at a few dozens of sites at most. Adding a storage node is then a cumbersome operation that typically requires the manual data redistribution. See the manuals of DB2 DBMS offering hashing partitioning and, e.g., of Non-Stop SQL for range partitioning.

Scalability and high-performance, including the access to an external data repository, are major goals for a database, [2], [3], [5], [6], [7], [12]. To experiment with the merge of all these technologies, we have coupled single-site AMOS-II and SDDS into a scalable distributed system. AMOS-II is for SDDS an application among others at the client site. It provides its object-relational declarative language AMOSQL to the applications. SDDS serves to AMOS-II as an external scalable RAM data storage and access manager. Data partitioning and its dynamic evolution are hidden from the users of AMOS-II. The database may reach sizes much larger than for a single site AMOS-II.

We have termed the prototype *AMOS-SDDS*. The coupling architecture on the client site should make AMOS-II and the SDDS manager capable to efficiently interface. This implies efficient function shipping, as well as effective return of possibly a lot of data retrieved by parallel queries. The SDDS server should in turn be made capable to evaluate the received functions. No SDDS prototype ever experimented with such capabilities.

Our solution is crucially founded on the new capability of an object-relational DBMS, with respect to a relational DBMS, usually termed *foreign* (external) *functions* (routines). A foreign function extends the basic capabilities of the DBMS and is accessible to the queries. AMOS-II was among first systems providing a foreign functions interface. It remains the only RAM DBMS with this capability, to the best of our knowledge.

In AMOS-SDDS, the *client* AMOS-II interfaces the SDDS services, i.e., of the SDDS client, through foreign functions. The SDDS client ships out the functions produced for the servers. To evaluate those, each SDDS server uses at its site also an AMOS-II, termed *server* AMOS-II. That one also uses the foreign function interface. It requests through those the services of the SDDS server (i) to get the local data and (ii) to ship back the filtered results to the client.

Below, we present the architecture of AMOS-SDDS. We describe the processing of the queries and the coupling technology between the SDDS clients and servers and the AMOSes. We show experimental performance measures, using queries with selections, projections, joins and aggregate functions as the benchmark. The join queries over distributed data are especially well-known as difficult for the efficient processing, [1], [14]. The experiments refine basic design issues that do not seem decidable on through the theoretical analysis alone.

The results prove the validity of our design. AMOS-SDDS appears very efficient. In particular, it may process faster volumes of data that AMOS-II alone could still handle. The design choices that appear should also help the future technology of the scalable high-performance DBMSs in general.

Section 2 recalls principles of an SDDS. Section 3 presents AMOS-II. Section 4 introduces the AMOS-SDDS architecture. Section 5 presents the performance study. Section 6 concludes the paper and presents future research directions.

2 Scalable Distributed Data Structures

An SDDS is a collection of records constituted each from a key and some non-key data. Records are stored at SDDS servers. The server's storage space is called a *bucket*. Buckets and servers are numbered 0,1… The SDDS is initiated as bucket 0. Inserts that overflow the file trigger bucket splits. Each splits appends a new bucket that receives about half of records from the split bucket.

An application searches, inserts and updates SDDS records from the SDDS *client* site. To address the servers, each client has the *image* of the actual file. The image maps each key to a bucket address, typically through a linear hash function or an index. The image may also contain a multicast address shared by all the buckets. This address is especially for the queries to non-key data. Such queries translate typically to parallel scans.

Initially, the image contains bucket 0 only. A current image may be inaccurate, as SDDS splits are not posted synchronously to the clients. A client may send a key search or an insert etc. to an incorrect server. Each server checks through its *checking* algorithm whether it is the correct one for the received query. If not, it forwards the query to another server determined though the *forwarding* algorithm. The correct server that finally receives the query sends back to the client the *Image Adjustment Message* (IAM). The IAM allows the client to adjust its image so that at least the addressing error that triggered the IAM does not get repeated. The IAMs make images to follow the evolution of the file state, less or more adequately.

The client that sends out a parallel scan, through multicast instead of a series of unicast messages, ends it up through some *termination* protocol. A *probabilistic* termination occurs when no reply message comes after some reasonable time-out. One tunes the time-out to reasonably guarantee that all servers that should reply do it. This depends on the SDDS size, server and network speeds etc. A *deterministic* termination occurs in contrast only when all the servers that should reply did it. The deterministic termination protocols are SDDS specific.

For AMOS-SDDS prototyping, we use the RP* scheme for scalable distributed range partitioning. Like in a B-tree, records in an RP* file are lexicographically ordered according to their keys. RP* supports efficiently the range queries. Each bucket has its *range* defined in its header by two values l and L called the *minimal key* and the *maximum key*. A bucket may contain key c iff $l < c \leq L$. Bucket 0 has the initial range of $(-\infty, +\infty)$. It splits into two buckets when the number of records to store exceeds the bucket *capacity* of $b >> 1$ records. Bucket 1 is then appended and receives the higher half of the bucket 0. This process iterates for every bucket that overflows when the file scales. At any split, if $\boldsymbol{c}$ denotes the corresponding median key, after the split the range $(l, L]$ of split bucket decreases to $(l, \boldsymbol{c}]$, while new bucket gets range $(\boldsymbol{c}, L]$. This process creates and maintains the range partitioning.

3 AMOS-II DBMS

AMOS-II is a distributed RAM multi-database system where one can choose between setting up wrappers for external data sources or storing data locally in the RAM databases. For this experiment, we only use the RAM storage manager and OO query processor of AMOS-II, i.e., the single site AMOS-II configuration [15], [17]. AMOS-II offers a declarative query language AMOSQL that can be embedded into C, Java, and Lisp. AMOSQL uses the object-relational paradigm where data are objects whose values are functions. A relational table typically correspond to an object whose OID is the key and attributes are function values. An external program interfaces AMOS-II using the *call-level functions* in two ways:

- The *callin* interface. Especially, the *a_execute* function dynamically executes an AMOSQL query.

- The *callout* interface. AMOS-II calls in this way the foreign functions (routines). The *a_emit* function allows the external program to pass the results (tuples) back to AMOS-II for further query processing or storage.

Foreign functions through the *callout* interface extend the manipulation capabilities of AMOSQL for specific user needs. One can develop these functions in C, Java, or Lisp.

4 AMOS-SDDS Coupling Architecture

We refer to the coupled AMOS-II and SDDS client as AMOS-SDDS *client*. Likewise, an SDDS server coupled with server AMOS-II becomes the AMOS-SDDS *server*. Figure 1 presents the overall AMOS-SDDS coupling architecture. Figure 2 shows the query processing steps.

SDDS clients and servers constitute for AMOS-SDDS clients and servers a scalable distributed communication platform. These handle all the messaging and data exchanges between the sites. The SDDS servers constitute the scalable distributed data storage. The user or application calls AMOS-SDDS at the client site. An AMOS-SDDS query can be an SDDS query requesting some records, e.g. an RP* range search. SDDS client processes such queries directly. Alternatively, the AMOS-SDDS query can be an AMOSQL query. The user specifies a query interactively. The application uses the *callin* interface through the *a_execute* function. These queries go to the client AMOS-II. A query may address local data cached on client AMOS-II or external data in an SDDS file seen then as some AMOS-II data. A record basically corresponds to one tuple whose OID is the RP* record key. The function values constitute the other attributes stored as non-key fields in some internal format.

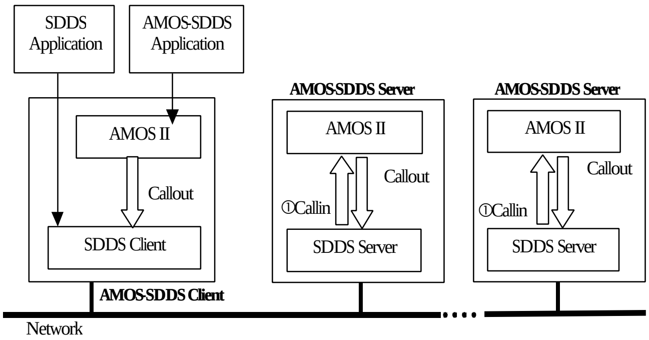

Figure 1. AMOS-SDDS Overall Architecture

The SDDS file is addressed through foreign functions and the *callout* interface. The user is aware that some data to address are external to local AMOS-II. The query formulation does not depend, however, on their actual distribution and scalability. Those are transparent to AMOS-SDDS user.

A *ship* function is a foreign function to external data, an SDDS file specifically. It ships some function (query and/or parameters) to servers. This is an SDDS query with, perhaps, an AMOSQL query within. The SDDS client expedites any such query. The server uses the AMOSQL query, if present, to locally filter the data. One invocation of a ship function at the client may loop over several function shipping ’s to the servers and returns of partial results to the client.

The client sends out an AMOS-SDDS function with an AMOS-II query through a procedure termed *Send-Amos* and that with an SDDS query through *Send-SDDS*, Figure **2**. They communicate with the server through different communication ports. The messages basically use unicast or multicast UDP messaging. An AMOS-SDDS server that receives an SDDS query processes it by the SDDS server as usual. Upon receiving an AMOSQL function in contrast, the server locally calls-in the server AMOS-II. The server AMOS-II internally calls-out the SDDS server again, perhaps multiple times, for the local bucket scan and record delivery. It uses for this purpose again specifically designed server foreign functions. The server AMOS-II filters the records. It returns those that satisfy the query to the SDDS server for the return to the client. The result is assembled in an AMOS-II *scan*, which can contain several tuples. The SDDS server copies with some reformatting the results in *scan* into its communication buffers. It then sends the buffers to the SDDS client using a TCP connection.

The SDDS client assembles the records received from the servers. It performs the deterministic termination protocols to detect the end of the data shipping. It also extracts the values in the records. These are pipelined back to client AMOS-II using the *a_emit* function. The pipelining occurs simultaneously to the reception processing. Client AMOS-II performs eventually *post processing* and makes the selected data available to the application.

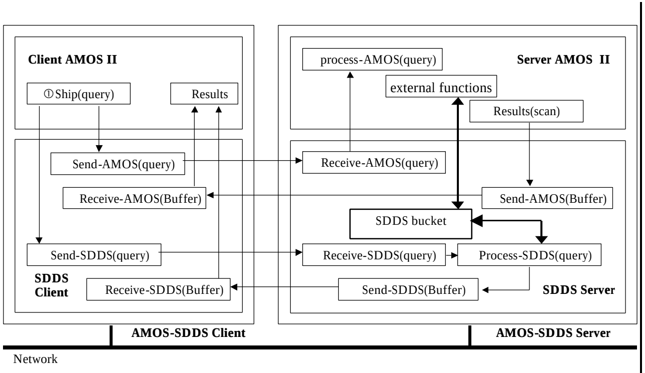

Figure 2. AMOS-SDDS Query Processing

As it appears, the AMOS-SDDS clients and servers use SDDS clients and servers as scalable distributed communication platform. The SDDS servers constitute also the scalable distributed data storage. Both roles require AMOS-SDDS client to decompose the AMOSQL query into subqueries suitable for the scalable distributed processing. This includes especially the efficient data communication and parallel use of the servers. At the client, the subqueries are basically managed by the ship functions. There are typically many ways to decompose a query in this way. Likewise, there are many ways to set up the execution of a subquery at the AMOS-SDDS servers. Finally, there are numerous choices for the communication. All these choices constitute the core issues for the AMOS-SDDS design. They especially concern the ship functions, the server foreign functions, and its overall call-in algorithmic. These are obviously complex issues. We discuss below those we have faced for the foreign functions implementing the benchmark query.

5 Performance analysis

5.1 *Rationale*

The implementation of the AMOS-SDDS required several design choices. To review some, first, at the communication level, one should decide whether to pipeline records between the sites as soon as produced, or to buffer them to send several in a single message. The latter issue opens that of best buffer sizes. Both issues, but especially the former, often open also that of choice between UDP and TCP/IP messaging. Next, the record format for the tuples communication should be chosen, especially one has to decide whether to use variable length fields. If the servers differ by their CPU speeds, the bucket sizes should differ for the load balancing as well. If the query contains a distributed join, one has to choose a strategy for its evaluation. A distributed nested loop join subquery may ship at once several (how many?) values of the join attribute for the bucket scan. It may, or may not, be effective to sort the values to send out at the client. One trades here the halving of the server scan time against the additional sort time at the client. Also, one may use either the deterministic or the probabilistic termination for the scans. For the latter, one should optimize the timeout.

At the server side, one should architect the AMOS-II subquery processing. It generally divides between the server AMOS-II and the AMOS-SDDS server, especially within the foreign functions. Many choices exist in particular for distributed joins. The comparison between the tuples gotten from the client and those local to the server for a distributed nested loop evaluation, may be done on the one hand entirely by the foreign function. Another strategy is to import all the tuples received and stored locally into server AMOS-II and then perform the join by this system. This strategy implies additional importation cost, but may be more efficient anyhow, since AMOS-II is optimized for join processing. In particular, the importation may create an index on the join attribute, speeding then up the computation.

It did not appear generally that the discussed and other related design choices could be made on the theoretical basis only. Furthermore, it did not seem possible to determine in that way, whatever could be the choice, the overall efficiency of AMOS-SDDS system. This concerns especially the scalability analysis. Both its components AMOS-II and SDDS are highly complex software. The coupling obviously adds-on on that. In particular, there does not seem to be any easy theoretical way to measure the efficiency of the *call-in* and *call-out* coupling. In brief, the experimental performance analysis of such systems is a must.

Probably best approach to the experimental performance determination is to use some complex benchmark data sets and queries. While many are well known, it does not seem any benchmark exist yet for our purpose. We have designed therefore a suitable benchmark on our own. The goal was to realize reasonably complex typical database operations. We have designed the queries with a quite selective join described below. We have heavily experimented the system performance using the benchmark under various conditions.

5.2 *Benchmark data set and queries*

The benchmark data were tuples in table **Person** (**ssn, name, city**). The ssn values are consecutive integers from 1 to 300,000. Name and City are random char strings of variable length. We have populated the table with 20,000 to 300,000 tuples, depending on the experiment. For the reference performance measures below concerning AMOS-II alone, the table was created entirely within this system. For the experiments with AMOS-SDDS it was set up as an RP* file at one or more servers. The **ssn** value was used as the partitioning key. The record size was then 25 bytes on the average.

The benchmark queries basically requested couples of persons living in the same city. For the partitioned RP* file, records of such persons were likely to be at different buckets, as the distribution of the records according to the city or name was random. There were fifty different cities generated so that the join selectivity, i.e., the ratio of the number of selected tuples to the size of the Cartesian product, was about 1.6 %.

On one hand, we have formulated and run our query, termed for this purpose *Query 1*, using AMOS-II alone with all the data. Its AMOSQL formulation was:

(Query 1)
select ssn, ssn1
from integer ssn, integer ssn1 character name, character name1 **where** person(ssn)=<name, city> and person(ssn1)=<name1, city> and ssn<ssn1;

The join was evaluated using the nested loop or an index on **city**. For AMOS-SDDS, the benchmark query with foreign functions, termed *Query 2,* was formulated as follow:

(Query 2)
select ssn, ssn1
from integer ssn, integer ssn1, character buffer
where sdds_fullextent()=buffer and f_ship(buffer)=<ssn, ssn1>;

Experiments with the execution strategies of Query 2 evaluated various design issues outlined above. The join calculus generally used the distributed nested loop. Details of the algorithmic and conditions of the execution, varied with the experiments.

First experiments used a query processing strategy called *E-Strategy*. The name recalls that we constructed the join tuples on the servers externally to AMOS-II, entirely within the foreign functions. It appeared that *E-Strategy* required a quite skillful programming. We have therefore experimented also with an alternative implementation design of the benchmark query Query 2, we call below *I-Strategy*. This one imports the bucket content and each buffer when it comes into the server AMOS-II. The latter performs its own query evaluation.

I-Strategy is potentially simpler to implement and more extensible than E-Strategy. It allows reusing join capabilities of AMOS-II, instead of reprogramming them in the foreign functions. Especially, it more easily allows for index lookup joins. The local indexes can be built at each server during the importation process using standard AMOS-II capabilities. The result for join queries shows an important performance advantage of I-Strategy. For 5 servers, the ratio is 6 times for the nested loop, and 9 times when the index is built! Unlike for our join queries, E-strategy is a clear winner for aggregate functions Count and Max. The ratio reaches almost 19 times for the counting! The obvious reason is the time spent by I-strategy on the importation. The use of external functions may thus be highly advantageous for certain kinds of queries. Details of measurement campaign are in [13].

New queries are also potentially easier to accommodate. Each server AMOS-II may handle entirely and most efficiently their local decomposition and execution, as long as they invoke any usual internal functions. The basic potential drawback of I-Strategy with respect to E-Strategy is of course the additional importation cost.

5.3 Experiments platform

We experimented with AMOS-SDDS on a multicomputer we called *faster platform.* That platform consisted of six Pentium III 700MHz with 256 MB of RAM running Windows 2000 on a 100Mbit/s Ethernet network. One site was used as client and the five other as servers.

Below we first present experiments with AMOS-II alone. I-Strategy follows, with its implementation details. We evaluate it first on a 20,000-tuple file. The favourable result makes us evaluate I-strategy for a five times larger 100,000-tuple file. Then, we scale the basic 20,000-tuple file on more AMOS-SDDS servers, up to 15. Last we study the scale-up characteristics of AMOS-SDDS on a file that scales up to 300,000 tuples.

5.3.1 Experiments with AMOS-II

AMOS-II at first executed the join in Query 1, using the nested loop. Next the index on the join attribute was created. Query 1 was executed again. This made AMOS-II to lookup the index for the join. The result size of the join query is 3,990,070 tuples. The resulting performance was as follows:

	Elapsed time(s)	Time per tuple (ms)
Nested-loop	**263**	**13.15**
Index lookup	**45**	**2.25**

Table 1. Elapsed time of Query 1 for the 20,000 record file

Next, we have scaled the file with 100,000 records. There were fifty different cities generated. The result of the join query was now 99,951,670 tuples. Execution times were now:

	Elapsed time(s)	Time per tuple (ms)
Nested-loop	**6,557**	**65.57**
Index lookup	**1,181**	**11.81**

Table 2. Elapsed time of Query 1 for the 100,000 record file

The speed of the index join, almost seven times faster here than the nested loop join, matches the theory.

5.3.2 I-Strategy

5.3.2.1 Foreign Functions

The foreign functions on client and servers were redesigned for I-Strategy. Query 2 was formulated as before, except for new foreign function at the client **Sdds_fullextent2()**:

<u>(Query 2 for I-strategy)</u>

select ssn, ssn1
from integer ssn, integer ssn1, character buffer
where sdds_fullextent2()=buffer and f_ship(buffer)=<ssn, ssn1>;

Sdds_fullextent2() reads all the tuples of **Person** into an unsorted buffer and of 2,000 records. That buffer size appeared experimentally optimal for I-Strategy, and not exceeding the maximum length of an UDP message. Sorting did not appear useful neither for the bucket importation speed, although potentially it could requests all the tuples of **Person**. The records successively received, are repacked into buffers.

At each AMOS-SDDS server, when it receives the query and the first buffer, it invokes once a new function **Load_bucket()**. That one imports the local RP* bucket into the server AMOS-II internal table named also **Person**, (a stored function more precisely). The foreign function **Import_tuples()** is then invoked for each incoming buffer to import it into AMOS-II table **Person_Temp**. The importation may create the index on the join attribute on **Person**. These are local indexes at each server. The foreign function **AllSameCity2()** computes then the join between both tables, avoiding the duplicates:

Function AllSameCity2 ()-> integer ssn, integer ssn2 as
select ssn, ssn2 **from** character name, character name2, character city, integer ssn, integer ssn2
where person(ssn) = <name, city> and person_temp(ssn2)=<name2, city> and ssn<ssn2;

Once this join is computed, the AMOS-SDDS server sends the partial results to the client. It also calls the internal AMOS-II function **Clear_function()** to empty the content of **Person_Temp** in one call. This avoids costly erasure of the tuples one by one. AMOS-SDDS is then ready for next buffer from the client. After, the last, **Clear_function()** empties the internal table **Person**.

To evaluate the transmission time of the tuples produced by Query 2 from the servers to the client, we have created on the servers, the foreign function **count_AllSameCity()** that sends only the count of the result at each server instead of the whole tuples. **Count_AllSameCity()** is defined in AMOSQL as follow:

Function count_AllSameCity()-> integer size
as **select count**(**select** ssn, ssn2 **from** character name, character name2, character city, integer ssn, integer ssn2
where person(ssn) = <name, city> and person_temp(ssn2)=<name2, city> and ssn<ssn2) ;

5.3.2.2 Experiments with I-Strategy

Table 3 and Table 4 present the performance of Query 2 for I-Strategy. The file has 20,000 tuples distributed over 1 to 5 servers. The join is computed first through the nested loop, then through the index.

Server nodes	1	2	3	4	5
Nested-loop (s)	**128**	**78**	**64**	**55**	**48**
Index lookup (s)	**60**	**39**	**37**	**36**	**32**

Table 3. I-Strategy for Query 2: elapsed time

Server nodes	1	2	3	4	5
Nested-loop (ms)	**6.4**	**3.9**	**3.2**	**2.7**	**2.4**
Index lookup (ms)	**3**	**1.9**	**1.8**	**1.8**	**1.6**

Table 4. I-Strategy for Query 2: time per tuple

To build the index at each server during the importation costs some time. Nevertheless it appears a worthwhile effort. The elapsed time decreases by half for two servers, and by 1.5 for five servers. The resulting time per tuple appears under 2 ms for the file on more than one server, reaching 1.6 ms for the file on 5 servers. While the speed up is sub-linear, the figures show a quite impressive efficiency of the whole system.

5.3.2.3 Scaling the file size

For this experience, we have created the file of 100,000 records, as for AMOS-II in Section 5.3.1. The file was however an RP* file over 5 servers. Table 5 lists the results.

	Elapsed time(s)	Time per tuple (ms)
Nested-loop	**1,000**	**10**
Index lookup	**691**	**6.91**

Table 5. Performance of Query 2 with I-Strategy

The comparison to Table 4 and Table 5 for the 20,000 record file, show a slightly better than linear scale-up for the nested loop. The time per tuple increases indeed from 2.4 to 10 ms, i.e., 4.2 times. Likewise, for the index join, the ratio is 4.3 times.

The comparison to AMOS-II results for the 20,000-tuple file in Table 1 shows also, hardly unexpectedly, a much better scale-up for AMOS-SDDS. Thus for the nested loop the AMOS-II elapsed time per tuple increases by factor of 5, from 13.15 to 65.57 ms. Likewise, for the index join, by factor of 4.8, from 2.25 to 11 ms. Notice that the index join time in Table 2 does not include the index creation time, unlike for Table 5.

5.3.2.4 Scaling the file size and the number of servers

Table 7 and Figure 4 present elapsed times of Query 2 on a file that scales from 20,000 to 300,000 records on several AMOS-SDDS servers. We run many AMOS-SDDS servers at the same machine when the file size exceeds 100,000 records. Each one contains 20,000 tuples with up to 3 AMOS-SDDS servers on the same machine, i.e., 15 servers in total. Table 6 shows the total number of servers and the number of servers per machine according to the file size.

The join calculus already uses the distributed nested loop with *I-Strategy*. This one imports the bucket content and each buffer when it comes into the server AMOS-II. It takes 3 seconds to import the bucket content and to create index on join attribute city for 20,000 records. AMOS-II performs its own query evaluation. To evaluate the transmission time of the tuples produced by Query 2 from the servers to the client, we use the foreign function **count_AllSameCity()** that sends only the count of the result at each server instead of the whole tuples.

The extrapolated time results from the elapsed times minus the transfer time, divided by the number of servers per machine.

# tuples	**20K**	**60K**	**100K**	**160K**	**200K**	**240K**	**300K**
# SDDS servers	1	3	5	8	10	12	15
# Machines	1	3	5	4	5	4	5
Server / Machine	1	1	1	2	2	3	3

Table 6. Number of servers according to the size of the file

Q1 = **AMOS-SDDS** join; **Q2** = **AMOS-SDDS** join with count.

# tuples	**20K**	**60K**	**100K**	**160K**	**200K**	**240K**	**300K**
# SDDS servers	1	3	5	8	10	12	15
Result size	4M	36M	100M	256M	400M	576M	900M
Q1 (s)	**61**	**301**	**684**	**1817**	**2555**	**3901**	**5564**
Q2 (s)	51	185	335	986	1270	2022	2624
Q1 w. extrap. (s)	61	301	684	1324	1920	2553	3815
Q2 w. extrap. (s)	51	185	335	498	640	681	881
AMOS-II (s)	**46**	**430**	**1201**	**3106**	**4824**	**6979**	**10 933**

Table 7. Elapsed time of join queries (extrapolated for AMOS-SDDS)

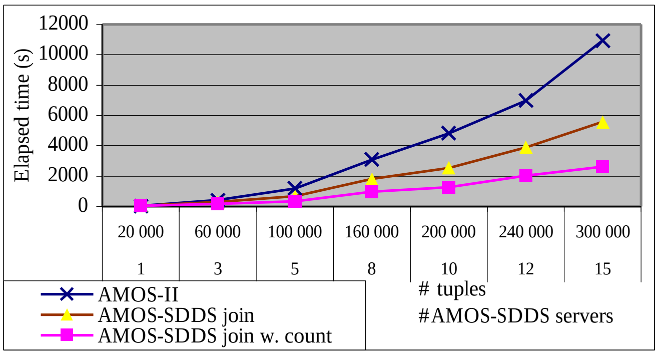

Figure 3 Actual measurements

Figure 3 compares the elapsed time of Query 1 to that of Query 2 in Table 7. The improvement ratio for the 300,000 record file is 1.96 times, i.e. by 49 %. Table 7 presents the extrapolation of the Query 2 execution times while running several AMOS-SDDS servers on distinct machines. The improvement ratio for the 300,000 record file is 2.86 times, i.e. by 65 %.

# tuples	**20K**	**60K**	**100K**	**160K**	**200K**	**240K**	**300K**
# SDDS servers	1	3	5	8	10	12	15
Q1 (ms)	**3,05**	**5,02**	**6,84**	**11,36**	**12,77**	**16,25**	**18,55**
Q2 (ms)	2,55	3,08	3,35	6,16	6,39	8,43	8,75
Q1 w. extrap. (ms)	3,05	5,02	6,84	8,28	9,6	10,64	12,72
Q2 w. extrap. (ms)	2,55	3,08	3,35	3,11	3,2	2,84	2,94
AMOS-II (ms)	**2,30**	**7,17**	**12,01**	**19,41**	**24,12**	**29,08**	**36,44**

Table 8. Time per tuple

Figure 4 shows the expected time per tuple of join query to AMOS-SDDS in Table 8.

The elapsed time of a distributed query can be divided into two parts: effective processing time over the servers and transfer time of the results from the servers to the client. The query processing is carried out in parallel on the servers and its duration depends on the number of distributed servers. Thus, the increase in the number of servers reduces the processing time of the same factor. The transfer time is however limited by the network bandwidth. The extrapolation of the elapsed time of the query with count shows a perfect scalability: Time per tuple remains constant when one increases the size of the file and the number of servers of the same factor.

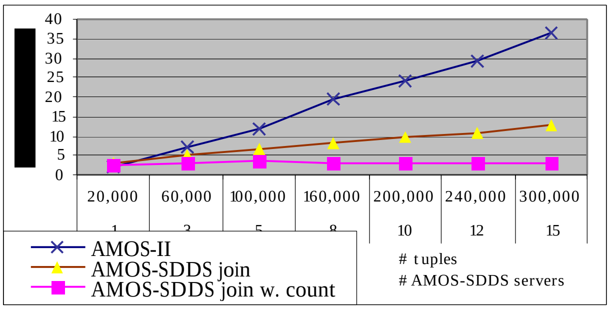

Figure 4 Expected time per tuple of join queries to AMOS-SDDS

6 Conclusion

AMOS-SDDS couples an SDDS with a high-performance DBMS, to improve the current technologies for high-performance databases and for the coupling with external data repositories. The experiments we have reported prove the efficiency of the system. The optimizations of various design choices, the I-strategy especially, decreased the file processing time for a join query from dozens of ms per tuple to possibly 1 ms per tuple. The AMOS-SDDS scalability that appeared from our experiments abolishes the cumbersome storage limits of a single site RAM DBMS technology. Its RAM query processing and scalable data partitioning are an improvement over the current parallel DBMSs technology. These capabilities should attract numerous applications, [2], [3], [5], [6], [7], [12].

The work on AMOS-SDDS continues. It includes deeper performance study of speed-up and scale-up characteristics. The performance of I-strategy that appeared brought further studies of AMOS-II as the sole bucket manager like in a parallel DBMS thus, except for the AMOS-SDDS capability of scalable partitioning. A related goal is a scalable distributed query optimizer for the AMOS-SDDS client.

Our coupling technique is not limited to AMOS-II. Other object-relational DBMSs supporting foreign functions can be potentially coupled to SDDS as well. We are currently studying the DB2 user defined table functions for this purpose. Likewise, we study the SQL Server for this purpose. The success may offer an attractive alternative to its static partitioning scheme.

Acknowledgements

This work was partly supported by a research grant from Microsoft Research. We thank Jim Gray, David Lomet, and Pierre-Yves Saintoyant for helpful discussions. Earlier work on SDDS was partly supported by grants from HP Laboratories, Palo Alto, CA, and from IBM Almaden Res. Cntr., San Jose, CA.

References

[1] Amin, M., Schneider, D., and Singh, V., An Adaptive, Load Balancing Parallel Join Algorithm. 6th International Conference on Management of Data, Bangalore, India, December, 1994.

[2] Comm. of ACM. Special Issue on high-performance Computing.(Oct. 1997).

[3] Clement, T., Yu, Meng, W. Principles of Database Query Processing for Advanced Applications. .Morgan Kaufmann, 1998.

[4] Diène, A. W. Litwin, W. Performance Measurements of RP*: A Scalable Distributed Data Structure for Range Partitioning. 2000 Intl. Conf. on Information Society in the 21st Century: Emerging Techn. and New Challenges. Aizu City, Japan, 2000.

[5] Gray, J. Super-Servers: Commodity Computer Clusters Pose a Software Challenge. Microsoft, 1996. (http://www.research.microsoft.com/)

[6] Grimshaw, A., Wulf, W. The Legion Vision of a Worldwide Virtual Computer. Comm. of ACM, Jan. 1997.

[7] Groff, J., R.,. & Weinberg Paul N.. SQL The complete Reference. Osborne/McGraw-Hill 1999

[8] Knuth, D. The Art of Computer Programming. 3rd Ed. Addison Wesley, 1998.

[9] Litwin, W. Menon, J., Risch, T., Schwarz Th. Design Issues For Scalable Availability LH* Schemes with Record Grouping. Distributed Data and Structures. Carleton Scientific, (publ.) 2000.

[10] Litwin, W., Neimat, M-A., Schneider, D. LH* : Linear Hashing for Distributed Files. ACM-SIGMOD Intl. Conf. On Management of Data, 1993.

[11] Litwin, W., Neimat, M-A., Schneider, D. RP* : A Family of Order-Preserving Scalable Distributed Data Structures. 20th Intl. Conf on Very Large Data Bases (VLDB), 1994.

[12] Musick, R., Critchlow, T. Practical Lessons in supporting Large-Scale Computational Science, , ACM Sigmod Record, December 1999.

[13] Ndiaye Yakham,Wane Diene, A., Litwin, W., Risch, Scalable Distributed Data Structures for Hight-Performance Databases, T.3-rd Intl. Workshop on Distributed Data Structures 2000, Italy.(See http://ceria.dauphine.fr/ceria.html)

[14] Ozsu, T., Valduriez, P. Principles of Distributed Database Systems. 2nd Ed. Prentice Hall, 1999.

[15] Risch, T. AMOS-II External Interfaces, Dept.. of Information Science, Uppsala University, Feb. 2000 (http://www.dis.uu.se/~udbl/amos/doc/external.pdf),

[16] Risch, T., Josifovski, V: Integrating Heterogeneous Overlapping Databases through Object-Oriented Transformations, Presented at 25th Intl. Conf. On Very Large Databases, Edinburgh, Scotland, September 1999

[17] Risch T., Josifovski V., Katchaounov T. AMOS II Concepts, Dept.. of Information Science, Uppsala University, Nov. 1999. (http://www.dis.uu.se/~udbl/amos/doc/amos_concepts.html)

[18] Seck, M.T., Ndiaye S., Diene A.W. Implémentation d ùne bibliothèque de primitives d àccès aux fichiers distribués et scalables RP*. CARI 1998.