The DASCOSA-DB Grid Database System

Jon Olav Hauglid, Norvald H. Ryeng, and Kjetil Nørvåg

Abstract Computational science applications performing distributed computations using grid networks are now emerging. These applications have new and demanding requirements for efficient query processing. In order to meet these requirements, we have developed the DASCOSA-DB distributed database system. In this chapter, a detailed overview of the architecture and implementation of DASCOSA-DB is given, as well as a description of novel features developed in order to better support typical data-intensive applications running on a grid system: fault-tolerant query processing, dynamic refragmentation, allocation and replication of data fragments, and distributed semantic caching.

1 Introduction

During the recent years, there has been a trend towards applications deployed on increasingly larger distributed systems with need for advanced data management. A prime example of such applications is computational science applications that uses advanced computing capabilities to understand and solve complex problems. Such applications frequently requires powerful computing resources, for example delivered through *grid computing services*.

While grid computing has gained maturity through the recent years, management of data in grid systems is less mature. Data storage and access is still mostly file oriented, and it is mostly left to users to manage files and their locations as needed. Although some support has emerged for metadata management, more advanced database features are not widely supported.

Department of Computer and Information Science, Norwegian University of Science and Technology (NTNU), 7491 Trondheim, Norway
{joh,ryeng,noervaag}@idi.ntnu.no
http://research.idi.ntnu.no/dascosa/

The goal of our research is a reliable *database grid*, with location-transparent storage, i.e., users/applications do not have to care about where data is stored and where queries are processed. The aim is sites cooperating on data storage and processing while retaining autonomy, i.e., a grid-wide database system. It is important to note how our context differs from more traditional approaches. The focus is on applications where large amounts of data is created and used on the same site, and where parts of the data, in particular summary data, are accessed by other grid participants.

An example of such applications is weather forecasting, where the national weather forecasting institutions have large amounts of locally collected data, do forecast, and make the resulting data available. They also store historical data. Both the summary data and historical data will be of interest to, and used by, other weather forecasting institutions and environmental researchers.

In this chapter we describe *DASCOSA-DB*, a distributed database system, which, in addition to providing location-transparent storage and querying, also includes novel features like efficient partial restart of queries and redistribution of query operators in the context of failure, dynamic refragmentation, allocation and replication of data fragments, and distributed semantic caching. A detailed overview of the architecture and the implementation of DASCOSA-DB is given, as well as a description of some of the features developed in order to better support typical data-intensive applications running on a grid.

The rest of this chapter is organized as follows: In Sect. 2 we give a short overview of other similar systems. In Sect. 3 we present the system architecture of DASCOSA-DB. Sect. 4 describes how data and metadata management is handled, and Sect. 5 explains query processing, including semantic caching and partial restart of failed queries. Our distributed monitoring and management tool is described in Sect. 6. An experimental evaluation of the system is provided in Sect. 7. Finally, we summarize our work and describe future research directions in Sect. 8.

2 Overview of Related Systems

Distributed databases and query processing is not a new field. For an introduction to distributed databases, we refer to [15]. A survey of distributed query processing is given in [12]. In this section, we will give an overview of systems that are similar to DASCOSA-DB. This includes both storage systems without query capabilities and query systems without storage capabilities, as well as complete database systems.

Much of the more recent work is based on peer-to-peer (P2P) networks, both unstructured and structured. Especially distributed hash tables (DHTs) have received much attention. A number of papers deal with focused issues such as query processing in DHT networks, including [2, 7].

OceanStore [13] is one of the storage systems without query capabilities. It provides an infrastructure for permanent storage and replication of objects, but no query system. Objects are accessed based only on their globally unique ID, and this ID has to be known in order to retrieve or update the object.

BigTable [5] is a large-scale distributed storage system with a model closer to relational databases. The storage model is similar to the relational model, but tuples are not stored or accessed as one unit. Instead, a row key and column key is used for both read and write operations. It does not provide more advanced query languages.

DASCOSA-DB does not provide its own storage infrastructure, but relies on an existing relational DBMS to store data. In that way, it is somewhat similar to the pure query engines that only provide a query processing service and no persistent storage.

Astrolabe [16] is one such system. Astrolabe is a distributed, hierarchical aggregation system designed for system monitoring. Astrolabe provides an interface that is similar to a database system, i.e., it provides SQL queries and standard database programming interfaces like ODBC and JDBC. To achieve scalability, updates are spread using a gossip protocol that guarantees eventual consistency. There is no guarantee that a client reads the most recent data, but if updates stop, all clients will eventually agree on the most recent value.

PIER [11] is a middleware query engine built on top of a DHT. PIER does not permanently store its data. Data sources publish their data in the DHT and update them regularly, and data that are not refreshed are removed. Typically, a PIER network will contain only object metadata (e.g., filenames, sizes, tags) and a reference to the original data object. Clients will query the network to get the references to the objects of interest and retrieve the objects separately.

The difference between these query engines and DASCOSA-DB is that, although DASCOSA-DB has a middleware architecture like PIER, it provides persistent storage by using a local database on each site. It is not necessary to constantly republish data, as is the case with PIER.

Among the systems that provide a full DBMS, with both query processing and storage, are Hyperion [17], Orchestra [22] and Piazza [6]. All these systems allow each site to have its own schema, and use schema mediation techniques to allow cross-site querying. PeerDB [14] also falls into this category of systems with heterogeneous schemas, but the approach to schema mediation is different. Instead of relying on schema mediators, information retrieval techniques are used to find matching relations.

DASCOSA-DB does not use schema mediation. The systems mentioned above are meant to connect existing databases and provide a common query interface. Although DASCOSA-DB is a distributed database system with a high degree of site autonomy, it still behaves as one system, not many different systems with a common interface.

Other systems based on a common schema include APPA [1], Mariposa [21] and ObjectGlobe [4]. APPA provides a multilayered solution on top of a structured or super-peer P2P network, where the bottom layer is a simple key/value-store and the top level provides advanced services such as schema management, replication and query processing.

Mariposa is a distributed database system that uses economic models to solve optimization problems. Mariposa sites buy and sell fragments and bid for the execution of queries. The trading and bidding makes sure queries are answered efficiently and that data are moved closer to where they are needed.

ObjectGlobe is a distributed query processing infrastructure that allows users to combine data sources and query operators from different providers at different sites to perform queries. Sites can sell data, query operators, computing power or a combination of these. The client combines these resources to a full query pipeline.

AmbientDB [3] is probably the system that bears the closest resemblance to DASCOSA-DB. AmbientDB is a system designed to provide full relational database functionality for stand-alone operation in autonomous devices that may be mobile and disconnected for long periods of time, while enabling them to cooperate in an ad hoc way with (many) other AmbientDB devices. A DHT is used both as a means for connection peers in a resilient way as well as supporting indexing of data.

Like AmbientDB, DASCOSA-DB is also constructed as a combination of middleware and federated databases, connecting the local databases of each site. The key difference is that AmbientDB is a system for mobile devices, which have low computational power and may frequently be disconnected from the network, while DASCOSA-DB is designed for sites that have the computational power necessary to do query processing and more stable network connections. DASCOSA-DB is also based on a DHT, like AmbientDB and PIER. However, the DHT is only used as a metadata catalog. Query processing uses point-to-point links following the query tree, more like Mariposa and ObjectGlobe. This is different from PIER, where the DHT is used extensively in query processing.

In terms of query capabilities, all sites of DASCOSA-DB are equal. There is no buying or selling of query operators or data. Data is fragmented, allocated and replicated according to the needs of the combined load of all sites, trying to keep the costs of network communication low. Query operators are shipped out to sites in order to minimize network costs by trying to perform most query operations on local data.

Many of the systems mentioned above support SQL-like querying and presents data similar to a normal relational database system. DASCOSA-DB is fully a relational database system that supports standard SQL.

A brief description of a DASCOSA-DB demonstration is given in [9].

3 System Architecture

In this section, the architecture of DASCOSA-DB is described. DASCOSA-DB consists of a number of autonomous sites connected to form a distributed database system. First is described how sites are connected, how data is distributed and how sites cooperate to execute queries and updates, and then the internal architecture of a single site is presented in more detail.

3.1 Distributed Architecture

DASCOSA-DB is designed as a middleware layer that binds together local DBMSs running on different sites to make a distributed DBMS providing location transparency. Fig. 1 shows the distributed architecture of DASCOSA-DB, as a middleware connecting local databases and applications to provide access to a large, distributed database. All sites are autonomous. There is no single site that controls the distributed DBMS. In this way, the sites act together as a peer-to-peer network.

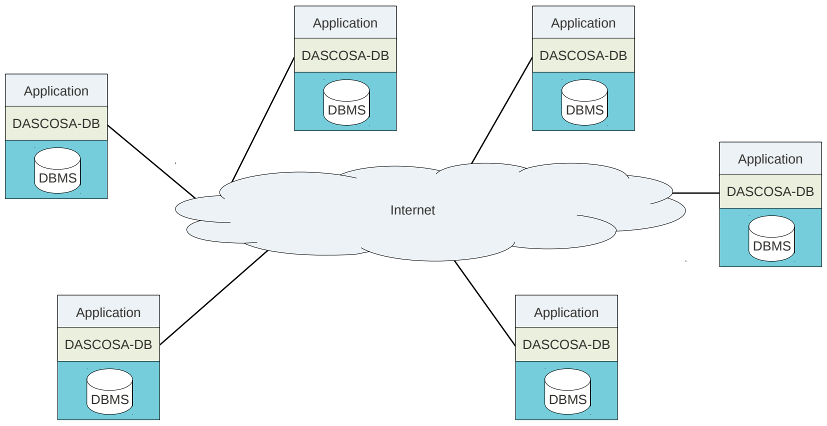


Fig. 1 Distributed architecture of DASCOSA-DB.

All sites connect to form a DHT. This DHT is used to store the distributed catalog, which contains information on all tables, table fragments, replicas and cache entries in the system. Currently, FreePastry[1] is used, but any other DHT may be used.

A new site wishing to join a running DASCOSA-DB system only needs to know the address of one connected site in order to join the DHT and thus be a part of the distributed database. When it has joined, it publishes information about its local metadata in the distributed catalog in order to make its local tables available to the rest of the system.

Sites communicate using messages. These messages can either be sent directly to a site if the address is known, or routed to the target site using the DHT routing mechanism. The latter method is used for catalog lookups and updates.

DASCOSA-DB supports the relational model and bases its storage on a local relational database management system. The current implementation uses JavaDB,[2] but any relational database management system may be used. The back-end database system can be chosen freely at each site.

[1] `http://freepastry.org/`

[2] `http://www.oracle.com/technetwork/java/javadb/overview/`

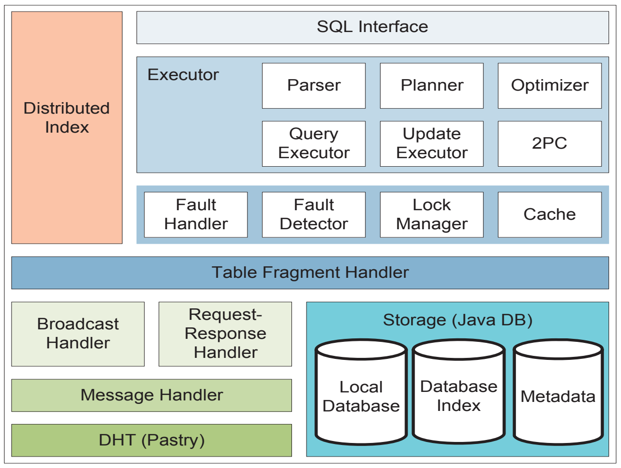


Fig. 2 High-level overview of the architecture of a DASCOSA-DB site.

The relational tables can be horizontally fragmented over a subset of the sites in the system. Each fragment can also be replicated. The distributed catalog maintains information about tables, fragments and their replicas. Creation and removal of fragments and replicas can be done using DASCOSA-DB's automated refragmentation method. Based on logging of read and write accesses, fragments can be split or joined, or replicas can be created and removed. This is done to reduce overall communication costs by making more data available locally where it is used and scale the number of replicas by the amount of writes. For example, a site doing heavy reads on a table fragment will get a local replica once this pattern is detected.

When executing queries, DASCOSA-DB utilizes query shipping. After query optimization, different query operators are allocated and distributed to sites in the system. This allows different operators to be executed by different sites in parallel. DASCOSA-DB also includes support for distributed semantic caching to speed up query execution. During updates, replicas are kept up to date using synchronous replication and transactions are handled using the two-phase commit protocol.

3.2 Site Architecture

The overall architecture of a DASCOSA-DB site is illustrated in Fig. 2. As described above, sites communicate using direct messages or using the DHT. Together with modules handling broadcasting of messages to the network and request-response pairs of messages, these constitute the communication subsystem in DASCOSA-DB.

Local storage on a site consists of three parts. First, there is the relational data. Relational tables can have one or more fragments and each fragment has one or more replicas. Therefore, the unit of local storage is a table fragment replica. The second part of local storage is the indices for these replicas. Finally, each site stores a part of the distributed metadata catalog. Which part of the catalog a given site stores, is determined by the distributed hashing algorithm and the site's position in the DHT.

Which replicas a site stores locally can change at runtime. Based on an analysis of logged reads and writes, the local Table Fragment Handler can dynamically decide to change the fragmentation and allocation of replicas in one of four ways:

- Coalesce two fragments into one fragment. This means that all replicas of both fragments will have to be altered.
- Split a fragment into two fragments. As with coalesce, this will have global effect for all replicas of the fragment.
- Send a copy of a local replica to another site so that this site can get its own local replica to speed up local accesses.
- Delete a local replica. This will reduce the effort needed to keep all replicas of a fragment up to date and will therefore make sense in periods with many updates.

The Fault Detector and Fault Handler are used to implement partial restart of failed queries. If a site detects that another site designated to execute a subquery has failed, it can handle this fault transparently from the rest of the query execution. This is done by relocating the failed subquery to other sites. In many cases, this can be done efficiently by not having the new sites restart the subquery completely, but rather continue where the failed site stopped.

Each site in the system can receive SQL queries and updates, for example using the provided user interface or using API calls. A received SQL statement is first parsed and transformed into relational algebra. If it is a query, a lookup in the distributed catalog is done to find location information about all involved tables. This information is then used by the Planner and Optimizer modules to generate a distributed query plan, including allocating the individual query operators to individual sites in the system. The operators are distributed to the involved site where the Query Execution module is responsible for the actual execution.

In order to facilitate easy interactive access to the system, as well as study configuration, distribution of data and query execution, DASCOSA-DB includes a monitoring tool that gives a live view of table fragments, replicas, catalog entries and cache entries. It also provides a live view of query execution, including network traffic and currently running query operators.

4 Distributed Data and Metadata Management

Tables in DASCOSA-DB may be horizontally fragmented based on the primary key, and DASCOSA-DB provides an adaptive fragmentation and replication system [10] that automatically moves data between sites as needed. In this section, the fragmentation process and then the replication of the fragments are described. Then it is described how metadata about fragments and replicas are retrieved from the local database when a site connects to the system and how it is published and subsequently retrieved from the global distributed catalog.

4.1 Fragmentation

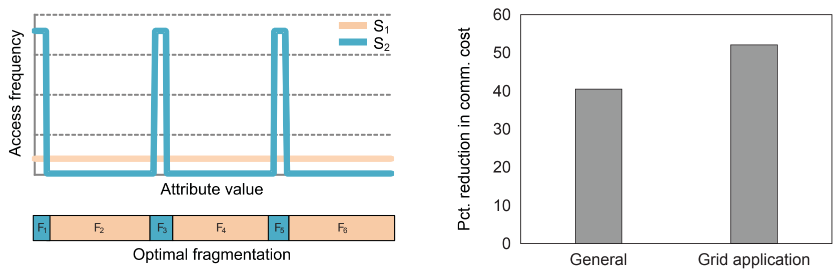
A table may be stored in its entirety on one site, or it can be fragmented over a number of sites. An unfragmented table is treated as a table having a single fragment. Tables are fragmented horizontally based on the primary key. Each fragment of a table is given a fragment value domain (FVD) that defines which range of the primary key domain has been allocated to the fragment. The fragments are non-overlapping, and the FVDs of all fragments of a table cover the whole primary key domain.

The FVD of a fragment may cover a much larger range than the range of actual tuples in the fragment. E.g., a newly created table consists of one fragment with the whole primary key domain as its FVD, even though it does not store any tuples yet. As tuples are inserted, updated, read and deleted, a larger part of the FVD is actually used, and the table may split into more fragments

The traditional way of fragmenting and replicating tables in distributed database systems has been to use fixed value ranges or rules defined by database administrators. In DASCOSA-DB, fragments and replicas are created and migrated automatically by the system to accommodate the current query load. Based on access pattern monitoring, DASCOSA-DB will try to keep the number of accesses to remote sites as low as possible. The FVDs and fragment placements are not fixed, so fragments can be split, coalesced and migrated automatically to adapt to changing workloads. Fig. 3(a) shows a simple example of how two sites with different access patterns access the same table. Site S_2 has a few hotspots, while site S_1 accesses the whole table uniformly and infrequently. In this case, DASCOSA-DB will split (or merge if the table is already split) the table into 6 fragments, $F_1, F_2, \ldots, F_6$. F_1, F_3 and F_5 will be allocated to site S_2, while F_2, F_4 and F_6 will be allocated to site S_1.

In order to make informed decisions about useful fragmentation and replica changes, future accesses have to be predicted. As with most online algorithms, predicting the future is based on knowledge of the past. In our approach, this means detecting access patterns, i.e., which sites are accessing which parts of which fragment. This is done by recording accesses in order to discover access patterns. Recording of accesses is a continuous process. Old data is periodically discarded so that statistics only include recent accesses. In this way, the system can adapt to changes in access patterns.

Given the available statistics, our algorithm examines accesses for each replica and evaluates possible refragmentations and reallocations based on recent history. The algorithm runs at given intervals, individually for each replica. Each site bases its decisions only on information available at that site, requiring no synchronization



(a) (b)

Fig. 3 (a) Access pattern and desired fragmentation. (b) Reduction in communication costs relative to static fragmentation.

with other sites. With master-copy based replication, all writes are made to the master replica before read replicas are updated. Therefore, write statistics are available at all sites with a replica of a given fragment. On the other hand, reads are only logged at the site where the accessed replica is located. This means that read statistics are spread throughout the system. In order to detect if a specific site has a read pattern that indicates that it should be given a replica, it is required that each site reads from a specific replica so that each site's read pattern is not distributed among several replicas.

There is a great potential for cost savings by improving fragmentation. Fig. 3(b) shows the reduction in number of tuples transferred over the network in DASCOSA-DB for two different workloads. In the general workload, all sites access tuples uniformly across a selected range of the whole table. 80% of the accesses are read accesses and 20% are write accesses. The reduction in tuple transfers is more than 40%. In the grid application workload, each site alternates between read phases and write phases, changing hotspots for each phase. The grid application workload has more clearly separated phases, and the savings are more than 50%. The results clearly show that the cost of splitting, migrating and replicating fragments pays off.

4.2 Replica Management

A table fragment is considered to be a logical entity. The physical entities stored in the local DBMSs are table fragment replicas. All fragments must therefore have at least one replica.

Replicas are kept up to date using synchronous replication. Every statement that changes the state of a fragment is sent to all sites with replicas. All replicas must be updated in order for a transaction to commit, and a two-phase commit protocol is used to ensure that all replicas agree on the decision to abort or commit the transaction.

Similar to the way fragments can be split or coalesced, replicas can be automatically created and deleted. Each site logs reads and updates to the locally stored replicas. A new replica is created at a given site if this site does a lot of reads. The idea is that the cost of transferring the replica to the site is less than having a constant stream of remote read requests. A local replica is deleted if there are few local accesses compared to the number of updates received. For both these mechanisms, the idea is to reduce the overall network traffic.

Not all replicas are treated equally. One replica is designated as the master replica. In order to ensure that automatic replica deletion does not delete all replicas, this replica is not eligible for deletion. The site containing the master replica has two special functions. First, it is the site where refragmentation decisions are made. This prevents two sites from independently and simultaneously deciding to, e.g., split the same fragment. Only the site with the master replica is able to do this. Second, the site with the master replica acts as a lock manager for the table fragment. This allows us to not have a centralized lock manager, which could become a bottleneck in a large system. When the system first boots, the catalog site storing the catalog entry for a table decides for each fragment of the table which replica becomes the master replica, and thus also which site becomes the master replica site. A new master replica site can be selected if the current master replica site crashes. It is also possible for the current master replica site to transfer this status in case of refragmentation.

4.3 Metadata Management

DASCOSA-DB uses a DHT to store and access the metadata catalog. The DHT provides a reliable and robust routing and lookup mechanism. Due to the DHT routing, catalog lookups are fault tolerant. The DHTs hashing function also distributes responsibility for metadata storage. All sites in the system participate in the DHT, and when a metadata object is published in the DHT, the DHT places it on one of the sites according to a hash of the object. Using a uniform hashing function, metadata objects are uniformly distributed among the catalog sites.

When a site joins the DHT, it scans its local database and inserts information on local objects into the DHT. Catalog objects will time out if they are not renewed, and sites periodically republish their information before the objects time out and are removed. This is done to ensure that erroneous information that may appear due to sites crashing after publishing their metadata is cleaned up regularly.

The catalog keeps track of tables and their schemas. For each table, it stores information about the primary key and the name and data type of all attributes. The catalog also keeps track of how tables are fragmented and replicated, i.e., how many fragments there are, the FVD of each fragment, and the number of replicas and their locations. Also, one replica of each fragment is designated the master replica, and the catalog stores this information.

The existence of caches of intermediate query results is also regularly published to the catalog in the same manner as table, fragment and replica information. For each cached query result, the catalog stores a semantic descriptor, location and timestamp. Information about cache entries is not looked up directly, but rather discovered as a side effect of table lookups. The cache lookup is included in table lookup requests and replies. This mechanism is described in more detail in Sect. 5.1.

5 Distributed Query Processing in DASCOSA-DB

DASCOSA-DB is a query shipping system where all sites store data and process queries. Queries may arrive from any site of the system, and the site that introduces a query to the system becomes the initiator site for that query. It is assumed that queries are written in some language that can be transformed into relational algebra operators, for example SQL.

5.1 Query Pipeline

A query enters the system at one site. This site, called the initiator site, becomes the coordinating site for this query. The initiator site transforms the query into an algebra tree. During query planning, the different algebra nodes are assigned to sites. This requires catalog lookups in order to transform logical table accesses into physical localization programs, e.g., a set of accesses to table fragment replicas. Sites can be assigned more than one algebra node so that one site can be assigned a whole subquery. As all sites have the capability to execute operators, sites storing table fragments used in the query are typically also assigned query operations on these fragments during planning. This tends to reduce network traffic as tuples can be processed locally. An example of an algebra tree with site assignment is shown in Fig. 4(a). The initiator site plays the role of coordinator for this query and executes an initiator algebra node that is the endpoint of the query result.

DASCOSA-DB can cache the intermediate and final results of queries. Each site autonomously caches results of locally executed queries and subqueries and registers these in the distributed catalog so that the caches can be found by other sites. These catalog entries contain a semantic description of the cached query result, the address of the site that stores the cache entry, and a timestamp used to check cache entry validity.

As Fig. 4(a) indicates, the complexity of a query increases with the height of the query tree. The query $T * U * V$ is more complex than $T * U$. If some of the intermediate results, like $T * U$, are cached, the more complex queries may be answered partly from these caches, saving both execution time and computational cost. More complex results in cache means larger savings when these caches are used. How-

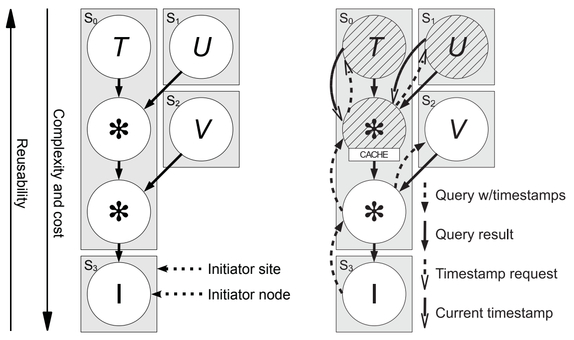


Fig. 4 (a) Example query plan. (b) Query dissemination with a cache hit.

ever, as the other arrow in Fig. 4(a) shows, the reusability of a result is higher for the less complex queries.

When a table is looked up in the catalog, the initiator site piggybacks a representation of the query to the lookup message. The catalog site that handles the lookup request sees this query representation and responds by piggybacking onto the response a list of suitable cache entries that might speed up query processing. Information about a cache entry is stored at the same site as one of the tables involved in the query that produced it. This means that after looking up all tables, the initiator site has been told about all caches involving the combination of these tables. During localization, the initiator site looks at these cache entries. If a relevant cache entry is found, the initiator site can rewrite the query to use the cache entry. This is done by including the query that produced the cached result as a subquery of current query and assigning the subquery to the site where it is cached.

After planning and possibly rewriting the query to use cached intermediate results, query dissemination begins by transmitting the algebra tree stepwise from the initiator site to the different sites involved. The root algebra node always stays at the initiator site. For each child of the root node, the initiator site sends out the subtree rooted at that child node to the child's assigned site. These sites, upon receiving query subtrees where the roots are assigned to them, loop through the children of the roots and ship them off to the sites to which they are allocated. This continues until all nodes have reached their destination. The result of this stepwise transmission is that each site knows the complete subquery for which it is the root.

However, if a site receives a subtree for a query it has in its cache, and if that cache entry is still valid, further dissemination of that subtree stops. Instead, the site prepares a special algebra node to produce the result from cache. To the sites higher up in the hierarchy, there is no way to tell if the result is served from cache or produced from scratch. This transparency allows the sites to make cache decisions without relying on central coordination. Fig. 4(b) shows query $T * U * V$ with a cache hit on subquery $T * U$. Site S_0 checks the timestamp of the cache entry against the timestamps of T and U to see if the cache is up to date. If it is, $T * U$ is delivered from cache, and the only query operator actually executed is the join of $T * U$ and V at site S_0. Site S_1 is never involved in the query processing, except when replying to the request for the timestamp of U.

Results of query operators are transferred between sites in tuple packets. The system supports stream-based processing of tuples, for example joins performed by pipelined hash-join [23]. This means that an algebra node usually can start producing tuples before all the tuples are available from its operand nodes. This makes it possible for nodes downstream to start processing as soon as possible and therefore lets more nodes execute in parallel. This requires each site to be able to accept and buffer yet unprocessed packets, but it allows data transfers to be made without explicit requests, thereby improving response time. In case of limited buffer availability, flow control is used to temporarily halt packet transmissions.

The result of any algebra operator is a candidate for caching at the site where it is produced. Sites are allowed to use any cache replacement algorithm they want. A cache entry is usable as soon as it is created, but in order to enable the query planners to plan on using cached results the cache entries must be registered in the distributed catalog. A site that has cached a result reports its existence to the same site that handles lookup request for one of the tables used to produce the result. E.g., if the cache entry is the result of $T * U$, the catalog stores the information about this entry at either the site that stores the catalog entry for T or the catalog entry for U. Any site that later looks up both T and U in order to perform a join is guaranteed to find this entry.

5.2 Standard Query Operators

DASCOSA-DB supports the typical query operators. At the lowest level, the scan operator accesses the local DBMS and delivers tuples of a table fragment. In order to speed up execution, special scan nodes exist that push selection and projection down into the local DBMS.

Selection and projection operators also exist to be inserted into the query tree when the operations cannot be pushed down into the local DBMSs. The selection operators also support set operators, i.e., IN and EXISTS, to compare against the result of subqueries.

The join operators include natural join, equijoin and outer join. These are implemented as pipelined hash joins. An operator also exists to produce the Cartesian product. Other operators include sorting, limiting, aggregation (including grouping), duplicate removal (UNIQUE) and a skyline operator.

All operators, except the scan operators, have flow controlled input and output streams with a general interface. This makes it possible to connect them in any meaningful way to represent a query. This generalized interface also makes it easy to ship queries around, since the input and output streams are network transparent.

For most normal cases in-memory operators suffice, but for large operand sizes there are also variants of these operators that will use disk to avoid excessive memory consumption.

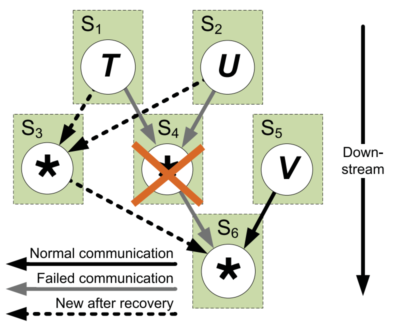
5.3 Fault-Tolerant Distributed Query Processing

The more sites that are involved in a query, the higher the probability of a site failing during query processing. Long queries and high churn rates in the system also increases the probability of site failures. The traditional way of handling failures focuses on update transactions, and the typical failure recovery is to do a complete restart of the failed transaction. Query failures have largely been overlooked. Complete query restart is an appropriate technique for small and medium-sized queries, however it can be expensive for very large queries, and in some application areas there can also be deadlines on results so that complete restarts should be avoided. In some cases, various checkpoint-restart techniques have been employed to avoid complete restarts of operations, but these techniques have been geared towards update/load operations, and in many cases implies that a query will be delayed until the failed site is back online.

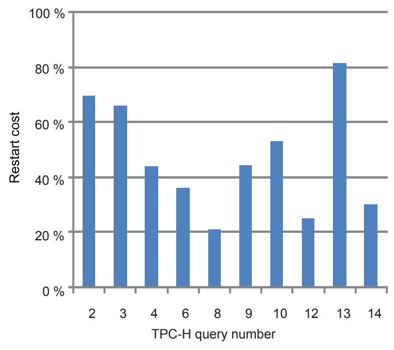
As an alternative to local checkpointing and complete restart, DASCOSA-DB supports partial restart of queries [8]. With partial restart, unfinished subqueries from failed sites can be resumed on new sites after failures. These restarted subqueries may also utilize partial results already produced before the failure — both results generated at non-failing sites and results from failing sites that have already been communicated to non-failing sites. The technique integrated in DASCOSA-DB can be compared to previous approaches like [20]. DASCOSA-DB's fault tolerant query processing 1) reduces query execution time compared to complete restart, 2) incurs minimal extra network traffic during recovery from query failure, 3) employs decentralized failure detection, 4) supports non-blocking operators, 5) handles recovery from multi-site failures, and 6) avoids duplicate tuples by deterministic delivery of tuples from base relations and operators. The query restart techniques can also be used to provide distributed suspend and restart of queries.

Fig. 5(a) shows a system executing the query $T * U * V$. Originally, only sites S_1, S_2, S_4, S_5 and S_6 are involved, but sometime during query processing S_4 fails. This is detected by site S_6, which is the recipient of the result of the failed algebra node. Site S_6 chooses S_3 to replace S_4, and reissues the query $T * U$ to this site. Site S_3 follows the normal query dissemination strategy and forwards the scan operators to sites S_1 and S_2. The particular challenges that have been solved in our approach relate to failure detection, selection of replacement site, and restart of the various relational algebra operators.

Failures during query processing are detected by using timeouts. There is no central failure detector. Instead, a site monitors all sites that produce the operands for query operators executing at that site. If a site failure is detected, a new site is selected for each of the failed operators. The impact of a failure is therefore localized



(a)



(b)

Fig. 5 (a) Example of query failure and restart. (b) Relative cost of restarted TPC-H queries.

— it only affects the sites receiving the results of the failed query operators. Other queries and subqueries executing at other sites continue as normal.

The replacement site selected to execute a failed query operator tries to pick off where the operator first failed. How this is done, depends on the operator. Two classes of operators can be identified: *stateless* and *stateful*. Stateless operators process tuples independently. Examples include projection and selection. For these operators, the number of operand tuples an operator has used to produce a given number of result tuples is stored. This number is transmitted with each packet of tuples sent in the network. Using this number, a replacement site knows where to start when resuming a failed operator. For example, assume that a failed site S_f was executing a selection. This selection was done on tuples received from another site S_o. The target site S_t for the selection, has received 500 result tuples when S_f fails. Assume that 800 tuples from S_o had been processed to produce those 500 result tuples. This fact will be known by S_t and transmitted to the replacement site S_r. S_r will then know that it should request S_o to resume sending tuples, skipping the first 800.

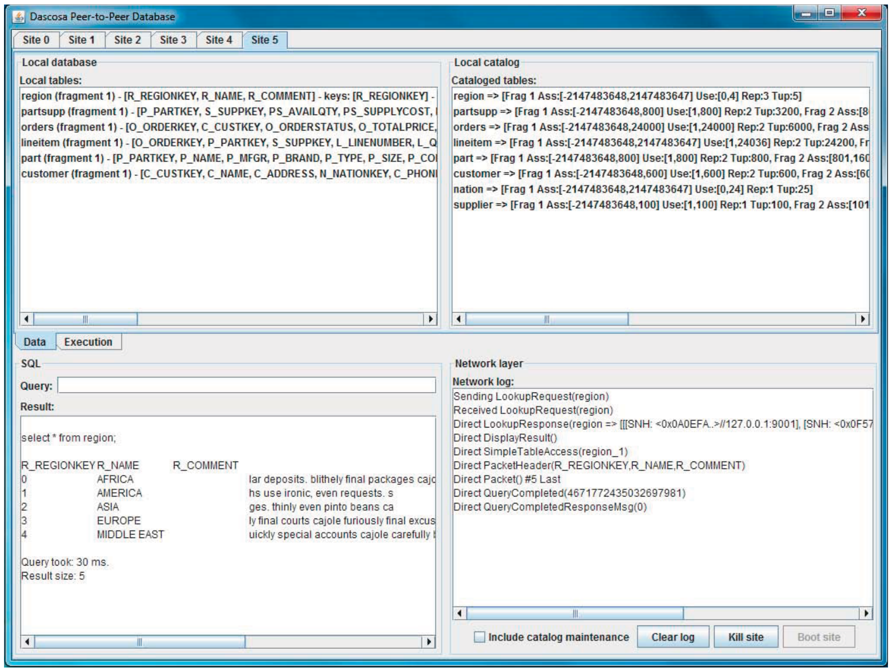
For stateful operators, on the other hand, each result tuple can be dependent on more than one operand tuple. Such operators include join and aggregation. When such operators are restarted, they must request operands to be replayed in full. However, they can still use the number of received operands before the failure to prevent sending duplicates. E.g., a join must get its two operands completely, but it can skip sending the first result tuples up to and including the number of tuples received from the failed site.

For this partial restart technique to work correctly, tuples must be produced by an operator in a deterministic order. Note that this does not mean that is has to be a sorted order. For scan operators, it is required that tuples are retrieved from the local DBMSs in a deterministic order. Further, it is required that other operators are deterministic so that they produce tuples in a deterministic order given the same ordering of operand tuples. Thus, this requirement reduces to having operators consuming tuples in a deterministic order. This is achieved by having operators consume packets of operand tuples in a round-robin order sorted on the ID of the source site of an operand tuple packet.

The results in Fig. 5(b) show the cost of a restart for a representative collection of ten TPC-H queries. The average restart cost is 50%. The two queries with the least gain (query 2 and 13) were also the two shortest queries. There is a constant overhead in detecting site failure and restarting queries. For the longer queries, this constant overhead is relatively small, so these queries have a lower restart cost.

6 Distributed Monitoring and System Management

DASCOSA-DB includes an integrated distributed monitoring and management tool. Fig. 6 shows the user interface which allows the user to issue SQL statements and monitor the state of the system in real time. It has proven very useful for the different research projects employing or extending DASCOSA-DB.



Local tables:

```
region (fragment 1) - [R_REGIONKEY, R_NAME, R_COMMENT] - keys: [R_REGIONKEY] -
partsupp (fragment 1) - [P_PARTKEY, S_SUPPKEY, PS_AVAILQTY, PS_SUPPLYCOST, I
orders (fragment 1) - [O_ORDERKEY, C_CUSTKEY, O_ORDERSTATUS, O_TOTALPRICE,
lineitem (fragment 1) - [O_ORDERKEY, P_PARTKEY, S_SUPPKEY, L_LINENUMBER, L_Q
part (fragment 1) - [P_PARTKEY, P_NAME, P_MFGR, P_BRAND, P_TYPE, P_SIZE, P_CO
customer (fragment 1) - [C_CUSTKEY, C_NAME, C_ADDRESS, N_NATIONKEY, C_PHON
```

Cataloged tables:

```
region => [Frag 1 Ass:[-2147483648,2147483647] Use:[0,4] Rep:3 Tup:5]
partsupp => [Frag 1 Ass:[-2147483648,800] Use:[1,800] Rep:2 Tup:3200, Frag 2 Ass:[8
orders => [Frag 1 Ass:[-2147483648,24000] Use:[1,24000] Rep:2 Tup:6000, Frag 2 Ass
lineitem => [Frag 1 Ass:[-2147483648,2147483647] Use:[1,24036] Rep:2 Tup:24200, Fr
part => [Frag 1 Ass:[-2147483648,800] Use:[1,800] Rep:2 Tup:800, Frag 2 Ass:[801,160
customer => [Frag 1 Ass:[-2147483648,600] Use:[1,600] Rep:2 Tup:600, Frag 2 Ass:[60
nation => [Frag 1 Ass:[-2147483648,2147483647] Use:[0,24] Rep:1 Tup:25]
supplier => [Frag 1 Ass:[-2147483648,100] Use:[1,100] Rep:1 Tup:100, Frag 2 Ass:[101
```

Query:

Result:

```
select * from region;

R_REGIONKEY R_NAME        R_COMMENT
0           AFRICA                      lar deposits. blithely final packages cajo
1           AMERICA                     hs use ironic, even requests. s
2           ASIA                        ges. thinly even pinto beans ca
3           EUROPE                      ly final courts cajole furiously final excus
4           MIDDLE EAST                 uickly special accounts cajole carefully

Query took: 30 ms.
Result size: 5
```

Network log:

```
Sending LookupRequest(region)
Received LookupRequest(region)
Direct LookupResponse(region => [[[SNH: <0x0A0EFA..>//127.0.0.1:9001], [SNH: <0x0F57
Direct DisplayResult()
Direct SimpleTableAccess(region_1)
Direct PacketHeader(R_REGIONKEY,R_NAME,R_COMMENT)
Direct Packet() #5 Last
Direct QueryCompleted(4671772435032697981)
Direct QueryCompletedResponseMsg(0)
```

Fig. 6 Screenshot from the DASCOSA-DB system monitoring tool.

DASCOSA-DB supports running more than one site on the same physical computer. All these sites will still communicate as if distributed and have separate local DBMSs. Running more than one site locally allows the user to easily examine the execution of distributed queries as the monitoring tool can observe all these sites.

The available views show which table fragments are stored at each site and the schema for each of these. The catalog view for a site shows catalog entries stored at that site. Tables are listed with the number of fragments and replicas, and each fragment entry shows the FVD, the actual used ranged and the number of tuples in the fragment. The catalog view also shows cached query results.

Network traffic monitoring is made easy by using the network log, which will list all messages received and sent by a selected site. This allows the user to, e.g., easily track the distributed execution of a query. Both query processing messages, catalog messages and other maintenance messages can be inspected.

The monitoring tool also allows the user to inspect running queries and follow the execution of algebra nodes as flow control changes the state of algebra nodes between processing and paused states. A complete view of all running queries and algebra nodes is provided.

Cache inspection is also provided. DASCOSA-DB has two caches: a restart cache that is used to provide fault tolerant query processing, and a semantic cache of intermediate query results. Each of these may be inspected through the management interface.

Finally, the management interface allows the user to simulate network failures and site crashes by toggling on or off message delivery to each site. When a site is disconnected, the rest of the system will notice its disappearance and adjust to the new situation. Queries involving the failed site will restart, and new master replicas will be appointed.

7 Experimental Evaluation

The individual features of DASCOSA-DB have been evaluated experimentally in earlier papers [8, 10, 18]. In this section, it is showed how the system, as a whole, scales. Evaluation of additional DASCOSA-DB features not described in this chapter can be found in [19].

7.1 Experimental Setup

The system consists of 10 interconnected sites running DASCOSA-DB. A TPC-H dataset is horizontally fragmented into five fragments. Each site stores one fragment, meaning that there are two replicas of each fragment. A set of 1000 random TPC-H queries with random values for substitution parameters is used. An 80/20 distribution is used both for query and parameter selection.

The number of sites that issue queries, and thereby the number of coordinator sites, is varied between 1, 5 and 10 to show how system performance increases with increased parallelism. Each querying site executes its queries in series, waiting for one to complete before issuing the next. The system is tested both with and without semantic caching enabled.

7.2 Results

The execution time of each experiment relative to a baseline is measured, where all queries were issued in sequence from a single site, without caching any query results. The results shown in Fig. 7 show that by increasing parallelism so that all sites issue queries, execution times are reduced by 25%. Since DASCOSA-DB allows queries to be issued from any site, the risk of the coordinator site becoming a bottleneck is reduced, and higher throughput can be achieved.

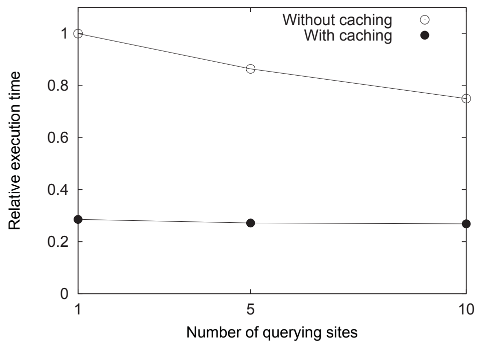


Fig. 7 Execution time relative to baseline.

Further, semantic caching reduces the run time with up to 73%. This considerable improvement is possible because parts of the algebra tree for a query is similar to some parts of other queries. These parts are reused to provide a quicker response to the query, freeing up resources that otherwise would be used to process each query from scratch.

The execution time does not decrease as much with increasing number of querying sites as was the case without caching. The reason for this is that there is not much more time to save after the reduction in execution time caused by semantic caching. Also, caching is a means to improve execution time of a series of queries, not parallel queries. The result has to be cached before it is used. Still, our semantic caching method makes it possible to reduce execution time of multiple parallel querying sites since cache entries are shared with all other sites.

8 Summary and Future Challenges

The central point of the grid is to present the user with readily available computational power without the need to know where this power comes from. This should also be the central point for data storage used by the grid, and our DASCOSA-DB is designed with this in mind.

We have presented a middleware system that transparently provides access to data distributed throughout the grid. Based on the relational model, our query shipping database system efficiently queries data in situ, while constantly adapting to the shifting workload by moving table fragment replicas closer to where they are used and by replicating data that has to be read by many sites. Semantic caching reduces the need to compute everything from scratch and allows new queries to take advantage of the intermediate results of queries that have already finished, even if they came from different sites. In case of failures during query processing, DASCOSA-DB will restart only the failed subquery. DASCOSA-DB also provides a distributed monitoring and management system.

Although we now have a working distributed database system, there is no lack of remaining challenges. More advanced optimization in the presence of cached data is needed. We will also study rank-aware operators which are important for many of the intended application areas.

Acknowledgments

The development of the DASCOSA-DB has been supported by grant #176894/V30 from the Norwegian Research Council.

References

1. Akbarinia, R., Martins, V., Pacitti, E., Valduriez, P.: Design and Implementation of Atlas P2P Architecture. In: Global Data Management, 1st edn. IOS Press (2006)
2. Bauer, D., Hurley, P., Pletka, R., Waldvogel, M.: Bringing efficient advanced queries to distributed hash tables. In: Proceedings of LCN (2004)
3. Boncz, P., Treijtel, C.: AmbientDB: relational query processing in a P2P network. In: Proceedings of DBISP2P (2003)
4. Braumandl, R., Keidl, M., Kemper, A., Kossmann, D., Kreutz, A., Seltzsam, S., Stocker, K.: ObjectGlobe: ubiquitous query processing on the Internet. VLDB Journal **10**(1), 48–71 (2001)
5. Chang et al., F.: Bigtable: A distributed storage system for structured data. In: Proceedings of OSDI (2006)
6. Halevy, A.Y., Ives, Z.G., Madhavan, J., Mork, P., Suciu, D., Tatarinov, I.: The Piazza peer data management system. IEEE Transactions on Knowledge and Data Engineering **16**(7), 787–798 (2004)
7. Harren, M., Hellerstein, J.M., Huebsch, R., Loo, B.T., Shenker, S., Stoica, I.: Complex queries in DHT-based peer-to-peer networks. In: Proceedings of IPTPS (2002)
8. Hauglid, J.O., Nørvåg, K.: PROQID: Partial restarts of queries in distributed databases. In: Proceedings of CIKM (2008)

9. Hauglid, J.O., Nørvåg, K., Ryeng, N.H.: Efficient and robust database support for data-intensive applications in dynamic environments. In: Proceedings of ICDE (2009)
10. Hauglid, J.O., Ryeng, N.H., Nørvåg, K.: DYFRAM: dynamic fragmentation and replica management in distributed databasesystems. Distributed and Parallel Databases **28**(2–3), 157–185 (2010)
11. Huebsch, R., Hellerstein, J.M., Lanham, N., Loo, B.T., Shenker, S., Stoica, I.: Querying the internet with PIER. In: Proceedings of VLDB (2003)
12. Kossmann, D.: The state of the art in distributed query processing. ACM Computing Surveys **32**(4), 422–469 (2000)
13. Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Wells, C., Zhao, B.: OceanStore: An architecture for global-scale persistent storage. In: Proceedings of ASPLOS (2000)
14. Ng, W.S., Ooi, B.C., Tan, K.L., Zhou, A.: PeerDB: A P2P-based system for distributed data sharing. In: Proceedings of ICDE (2003)
15. Özsu, M.T., Valduriez, P.: Principles of Distributed Database Systems. Prentice-Hall (1991)
16. van Renesse, R., Birman, K.P., Vogels, W.: Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. ACM Trans. Comput. Syst. **21**(2), 164–206 (2003)
17. Rodríguez-Gianolli et al., P.: Data sharing in the Hyperion peer database system. In: Proceedings of VLDB'2005 (2005)
18. Ryeng, N.H., Hauglid, J.O., Nørvåg, K.: Site-autonomous distributed semantic caching. In: Proceedings of SAC (2011)
19. Ryeng, N.H., Vlachou, A., Doulkeridis, C., Nørvåg, K.: Efficient distributed top-*k* query processing with caching. In: Proceedings of DASFAA (2011)
20. Smith, J., Watson, P.: Fault-tolerance in distributed query processing. In: Proceedings of IDEAS (2005)
21. Stonebraker et al., M.: Mariposa: A wide-area distributed database system. VLDB J. **5**(1), 48–63 (1996)
22. Taylor, N.E., Ives, Z.G.: Reliable storage and querying for collaborative data sharing systems. In: Proceedings of ICDE (2010)
23. Wilschut, A.N., Apers, P.M.G.: Dataflow query execution in a parallel main-memory environment. Distributed and Parallel Databases **1**(1), 103–128 (1993)