The Data Cyclotron: Juggling data and queries for a data warehouse audience
Goncalves, R.

Citation for published version (APA):
Pereira Gonçalves, R. A. (2013). The Data Cyclotron: Juggling data and queries for a data warehouse audience

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

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 7

Concluding remarks

This Dissertation presents the rise of a new approach for distributed data manipulation exploiting the new network hardware trends. It opens a vista on a research landscape of novel ways to implement distributed query processing. The research presented is a response to the call-to-arms in [92], which challenges the research community to explore novel architectures for distributed database processing.

In answer to such a call, the Data Cyclotron architecture was designed. It addresses the grand challenge of distributed query processing, come up with a self-organizing architecture which exploits all resources to manage the hot data set, minimize query response time, and maximize throughput without global co-ordination.

With a turbulent data movement through a storage ring built from distributed main memory and capitalizing the functionality offered by modern remote-DMA network facilities, the Data Cyclotron uses data movement between network nodes as an ally to improve system performance, flexibility, and query throughput.

The conceptualization of a full functional system was done with an harmonious integration of the Data Cyclotron with a DBMS, more precisely, a column-store. It is an outstanding solution for distributed data analysis on huge data warehouses. From multi-query parallelization to scalable processing of complex queries and from pulsating rings for dynamic ring size adjustments to the co-existence of multi-rings within a single cluster, the research paths to be explored are enormous. Therefore, in this Chapter we summarize the major contributions of this dissertation and the future research steps.
7.1 Contribution

With the recent developments in the research world to enforce new radical changes on the design of network topologies, the Data Cyclotron stands as one of few architectures without central coordination that has an elegant way to distribute and access data. At the same time, it has the grounds of flexibility to explore new and novel data analyze techniques using state-of-the-art hardware.

The Data Cyclotron has adopted the ring topology for a decentralized architecture where nodes share data or parallelize data computations without a central coordinator. Its simplicity is the key to explore new and un-orthodox algorithms for distributed parallel processing. Its communication pattern leverages the data routing latency and gets optimal bandwidth utilization at networks switches using simplified routing.

For a flexible integration with different applications, each Data Cyclotron node is defined by three layers, the network layer, DaCy layer, and application layer. Their interaction is achieved by a few support structures and routines having in mind an efficient platform for data loading, data forwarding, and data access by all nodes.

The Data Cyclotron delineation of its components leads to an architecture which can be integrated readily within an existing DBMS by injecting simple calls into the query execution plan. The performance penalty comes from waiting for parts of the hot-set to become available for local processing and the capability of the system to adjust quickly to changes in the workload set.

To achieve high throughput and low latency in the Data Cyclotron, the most relevant data for the current workload is identified and set with the highest priority to consume the available network bandwidth and storage ring space. Designated as hot-set, its decentralized management is responsible to assure fast access to the most relevant data and avoid data starvation for computations interested on data chunks with low relevance.

A ranking system is used to determine where a data chunk should be, i.e., in disk, memory, or storage ring. The ranking system is composed of ranking metric called the level of interest (LOI), and a threshold called the level of interest threshold (LOIT). LOI describes how popular was, is a data chunk for the current workload. A data chunk with low rank is more likely to be evicted then a data chunk with high rank.

Using a well known network simulator (NS-2), different approaches and methods were explored to reach the most efficient and robust solution to rank the data chunks popularity and define an optimal hot-set. With different scenarios, the results confirm our intuition that a storage ring based on the hot-set can achieve high throughput and low latency.

The experiments showed that a decentralized hot-set management can lead to an unfair management of less popular data when the storage ring is overloaded. To cir-
cumvent the problem a new state for the data was created, the warm-set. It contributes for a more precise definition of hot-set which is now refined to: a set composed of data chunks with the highest probability of utilization and optimal size for efficient network bandwidth usage, the highest throughput, and the lowest data access latency. Furthermore, an innovative cache management was designed to reduce latency for applications where instructions have dependencies and cannot be eligible for execution based only on input data from the storage ring.

The architecture (cf., Chapter 3) together with the hot-set management (cf., Chapter 4) were conceptualized into a full functional system, DaCyDB (cf., Section 5.2). DaCyDB is the integration of a MonetDB instance on each Data Cyclotron node. MonetDB was chosen for being a column-store which are known for being efficient for intensive data analysis, but also for using the operator-at-the-time paradigm and partitioned execution for efficient parallelism. DaCyDB is in favor of partitioned execution since it offers much better opportunities for speedup and scaleup then pipeline execution. By taking the large relational operators and partitioning their inputs and outputs, it is possible to use divide-and-conquer to turn one big job into many independent little ones and distribute them among nodes for processing. This is an ideal situation for speedup and scaleup [46].

The extended plans created for partitioned execution are used by DaCyDB to maximize the cooperative actions on the shared data chunks. Such cooperative work is only possible due to the strategy taken by MonetDB to unroll the loops through code repetition (cf., Section 5.2.3) which combined with code re-utilization, such as recycling [84, 83], turns the overhead of defining and executing long plans negligible. This cooperative access to the partitions together with the internal query parallelism contributes for high throughput and low query response time.

The DaCyDB validation and evaluation (cf., Section 5.3) was done in two different clusters for real workloads and different scenarios. The challenges and issues to cope with different bandwidths, traffic jams, un-balanced data distribution, and a flexible query parallelism to exploit a continuous data stream were identified and solutions proposed. Solutions to make DaCyDB more efficient for complex computations such as TPC-H queries. From a better plan optimization to an efficient management of intermediates to reduce virtual memory utilization, several improvements were considered and some tested with experiments. Furthermore, we delineated a new scheduler for an effortless and efficient intra and inter query parallelism.

At the same time, the protocols and routines presented for the hot-set management (cf., Chapter 4) were also validated. The evaluation was not intended to benchmark our novel architecture against contemporary approaches of single/parallel solutions such as MapReduce solutions. The engineering optimization space for the DaCyDB is largely unexplored and experience gained to date with the architecture would not do justice to
such a comparison. Furthermore, compared to MapReduce solutions, DaCyDB design is intended for a different data processing stage (cf., Section 6.9).

From the three data processing stages: data collection, data preparation, and data presentation [58], MapReduce frameworks were designed to be efficient on the data preparation stage while DaCyDB was designed for the data presentation stage (cf., Section 6.9.2). The MapReduce paradigm was designed for slow networks and exploit a distributed file system for fault tolerance. It provides data parallelism instead of process parallelism. It is tailored to scale out in a cluster of commodity machines. On the other hand, the DaCyDB uses fast network to share and access storage nodes. Its design is based on the new trends for data centers while system like Hadoop were designed for traditional data center composed of commodity machines. Nevertheless, both solutions can also be adapted to somehow be efficient on others data processing stages.

The DaCyDB efficient resource utilization and utilization of scale up and scale out as two complementary solutions turns DaCyDB into an efficient architecture for complex computations on hybrid clusters. Giving priority to in-memory processing and overlapping communication with data processing (a requirement to save energy [36]), it joins the green movement for data centers. At the same time, it aligns with the hardware trends for data centers such as diskless nodes.

The idea of data movement between network nodes is the key for all this flexibility and efficiency. The absence of static data allocation is exploited to explore different algorithms for distributed processing. Computations are split into sub-computations. They are sent to the ring and each of them settles on a different node following the basic procedures of a normal computation. They are processed concurrently and the individual intermediate results are then combined to form the final computation result. Such freedom gives the grounds for an application to scale out without be bounded to any complex scheduling algorithm or data distribution scheme.

Furthermore, a computation is not tied to be executed to any specific node or group of nodes. Instead, each computation searches a lightly loaded node to execute on; the data needed will pass by upon request. This way, the load is not spread based on data assignment, but purely on the node’s characteristics and on the storage ring load characteristics. This innovative and simple strategy intends to avoid hot spots that result from errors in the data allocation and query plan algorithms.

Based on the levels of utilization, each node decides to leave the ring or request a ring extension. This autonomous and dynamic adaption to accommodate the workload requirements is what defines pulsating rings. The pulsating rings provide an outlook for query processing workloads that can not be addressed with database sharding and map-reduce. It is a new concept to design fully flexible distributed query processing architectures. Moreover, it is conceivable that multiple pulsating rings live in the same
physical cluster. At any point in time, such a mesh consists of numerous overlapped pulsating rings. The model exploits the independence and autonomy of each individual where everyone, based on the workload flow, works for the same goal, high throughput.

7.2 Research directions

The Data Cyclotron opens a vista on a research landscape of novel ways to implement distributed query processing. Cross fertilization from distributed systems, hardware trends, and analytic modeling in ring-structured services seems prudent. Likewise, the query execution strategies, the algorithms underpinning the relational operators, the query optimization strategies and updates all require a thorough re-evaluation in this context.

In this Section we lay down the most promising research directions for the DaCyDB and the Data Cyclotron architecture. Starting with DaCyDB, an interesting direction is the integration of updates using a version scheme to support different levels of result accuracy, and thus allow a database to be queried while it is updated. To understand the potentials of DaCyDB architecture, we plan a performance evaluation on supercomputers to explore an aggressive solution to exploit iterative loading and diskless nodes as well as verify CPU affinity for efficient query parallelism. As last direction, the Chapter explains how easy is the integration of a generic row-store with the Data Cyclotron.

7.2.1 The space for updates.

An update can be classified into two categories, appends or records updates. For the first category, the append happens at the source nodes. One or more partitions are added to the source nodes and an update catalog request is sent around the ring by the node executing the update query. Once the request returns, the columns/table new version becomes official. The data is only loaded to the ring upon request using iterative loading (cf., Section 6.2.4). It is the most common situation for the application scenarios for which DaCyDB is targeted, i.e., the updates on data sets from eScience or data warehouses are mainly appends.

For a record update, the update is split into two phases, selection phase and update phase. The selection phase identifies the data chunks which contain the records to be updated. The selection statement is extracted from the update query to identify the data chunks. The data chunks are then requested to perform the update. A low granularity update to avoid the request of all partitions of a column.
An append of few records to each column is seen as a special record update. It avoids the creation of many partitions at the source nodes. In this case, the last partition is always the requested one. In case it is full, a new partition is created. For both cases, the update or a copy of the partition is sent to the source nodes to propagate the changes to the backups.

**Distribute updates.**

All update queries enter the ring at the same node. They are tagged with a timestamp to create a time order for all update messages. Their execution is however distributed. Hence, to have order preservation and data consistency DaCyDB uses database versioning. The versions control is done through a distributed lock. For each table a lock is put in circulation in the counter clockwise ring at the table’s creation time.

The update queries are picked first-in-first-out order by the nodes. The node who picks an update query for execution uses the timestamp to add a new version to the table’s lock. For each version a table’s lock is at one of three phases: selection phase (the data chunks were requested), update phase (the node is updating the data chunks), or done phase (the update succeeded and a new version was established). To speed up the first phase, an update request has more priority than a read request, therefore, a data chunk is loaded into the hot-set at the reception of an update request, i.e., it does not need more requests to be popular enough to enter the hot-set.

In the first phase, for each version, the data chunks IDs are added to the table’s lock. Like this, for disjoint updates over the same table, the update phase is done in parallel. However, simultaneous done phases per data chunks are not allowed. When the latter state is set, the node which has the updates for the next version on the same data chunks can proceed.

The node who updated the data chunks with version \( V \) is responsible to load the new version into the storage ring upon request. The data chunks are only allowed to be loaded once the lock with status done for version \( V \) has completed a full cycle, this is, all nodes are aware of version \( V \) and old copies from other nodes cache were invalidated. It is equivalent to a distributed commit of an update transaction.

Older versions are kept around and their metadata is stored in the tables locks and at the catalog of each node. Their maintenance is similar to the rest of the database, i.e., using its LOI each data chunk version bounces between the cold-set and hot-set. Once it expires it is dropped from the database.

In an ideal situation, a new version is defined and a logical copy is created without data duplication. In DaCyDB exists partial duplication. In case a data chunk \( Chk \) is modified, a new materialized view \( Chk' \) with the updates is created. Hence, there is
some data redundancy between the $Chk$ and $Chk'$. If not modified, the data chunk is used for several consecutive versions. It reduces the amount of resources to store several database versions. Furthermore, it reduces the number of data chunks flowing in the storage ring.

**Results accuracy.**

A database version represents a unique, consistent view of the database, distinguished from other versions by time and an unique identifier. In the context of DaCyDB, the user does not specify a specific version, it uses a time reference, or an interval, to define on which version should the query be executed. It is a light versioning distributed database compared to Hbase [73] or BigTable from Google [33].

With distributed updates, depending on the number of records to be updated, several read-only queries can start between the update announcement, i.e., its registration in the lock, and its commit. DaCyDB explores the updates timestamp to allow a user to define the result accuracy of his queries. The concept is used in bitemporal databases such as TimeDB [131].

A query at registration time is tagged with a timestamp and through a time reference indicates the accuracy of its result. For example, for query $Q$ the result is considered accurate if it uses the last stable version at its registration time. Based on this information, DaCyDB inspects the locks flowing around the ring to determine the table’s version $V_t$. $V_t$ is then used on the request calls to collect the correct data chunks version from the storage ring.

An optimistic update approach is also possible. Since the data chunks for each version are not allowed to be loaded until it reaches the done stage, DaCyDB can pick the latest version registered five minutes before $Q$ registration $V'$ and send the query for execution. In mean time, if the update for $V'$ is aborted, the query is restarted and a previous version is selected and the same steps are followed until a stable version is founded. Such approach allows the user to query a database while it is being updated.

A similar solution for centralized systems at the records level was presented in [111]. The work contains the recipes to build a transient versioning system. It contains methods for efficient and flexible transient versioning of records to avoid locking by read only queries. Like [20] it is a finer grain solution, i.e., they work at the record level. The same for the Broadcast Disks multiversioning model to handle updates [124], this is, it works at page level. The versioning model for DaCyDB works at data chunk level. It is intended for large data sets and distributed data processing. Nevertheless, the concepts and ideas described in [111, 124] should re-evaluated and re-considered for the DaCyDB context.
7.2.2 Supercomputers

The two clusters used for evaluation of DaCyDB (cf., Section 5.3.1) follow the standard design of modern clusters for distributed data analyses. They represent the near future of a cluster composed by commodity machines. The nodes characteristics have imposed some challenges and requested some conservative solutions to achieve low query response time and high throughput. The lack of memory for intermediates and the communication speed among the nodes required the definition of an efficient hot-set management to make sure the most relevant data is in circulation. What would happen if the resources available per machine are twenty time bigger, i.e., the dream cluster?

The dream cluster for DaCyDB would be a cluster composed by nodes with huge non-volatile main memories, such as STT-MRAM (spin-torque-transfer magnetoresistive random-access memory), network links reaching dozens of GB/sec to interconnect them and access independent storage nodes with an efficient iterative loading service as described in Section 6.2.4. Such luxury of resources would emphasize even more the DaCyDB advantages for distribute query processing.

To study the system behavior in such luxury conditions, to learn how to explore them and to design an aggressive architecture, we intend to evaluate DaCyDB in the Huygens cluster \(^1\) from our national computer center \(^2\). Each node in Huygens has 16 dual cores, resulting in 32 cores per node. 92 of them has 128 GB of memory (4 GB/core) and 16 nodes with 256 GB memory (8 GB/core). Each node has 2 HCA (host channel adapter) and each HCA has 4x DDR InfiniBand ports, this is, 160 Gb/sec (20 GB/sec) inter-node bandwidth.

**Pure in-memory data processing.**

Using a ring of 16 nodes, the ones with 256 GB main memory, we intend to study how DaCyDB would behave with diskless nodes, i.e., no cold-set. In this situation, there is only data resident in memory. For the hot-set management, the data instead of being unload to the cold set, it would simply be dropped. Once the data becomes relevant enough to be loaded to the hot-set, the data chunks would be requested from the source nodes exploiting the large network bandwidth and the iterative loading service.

The data chunks would be retrieved in parallel from the storage nodes (defined using the machines from the 92 nodes set), and their load to the ring would be spread among a sub-set of nodes to avoid tension on the buffers of a single node. The system would be split into two independent layers, the storage layer and the computational

---

\(^1\)https://www.sara.nl/systems/huygens

\(^2\)https://www.sara.nl
layer. A split to efficiently allocate specialized resources for I/O tasks or computational tasks.

The strategy is shared by distributed processing systems which retrieve data from the cloud such as Daytona\(^3\). Daytona is a kind of MapReduce framework from Microsoft for which the input data is stored in the cloud. It exploits a stream of data from Azure cloud service to feed the computational nodes which are running map and reduce tasks.

**New parallelism techniques.**

Another aspect to be explored in Huygens is the design or utilization of new techniques for parallel processing. Exploiting its SMP (Symmetric Multiprocessing) nodes we intend to explore different techniques, such as CPU affinity, to design new scheduling algorithms.

In a SMP node each core is split into two virtual CPUs, leading to 64 (virtual or logic) CPUs per node. With 256 GB of main memory, each virtual CPU gets 4 GB for input data and intermediates. To each of them, independent *DataFlow* blocks or sub-computations are scheduled for execution. A request from a sub-computation \(S_f\) makes the data chunk to be stored in the memory region assigned to the virtual CPU executing \(S_f\).

For CPU affinity, the sub-functions are scheduled using the MapReduce paradigm for multi-core machines such as Phoenix library [125] or Metis [106]. These libraries use CPU cores as nodes and schedule the computations in the same matter as MapReduce. Through data requests DaCyDB does not need an optimized structure, such as a hash with B+tree, to efficiently shuffle between cores, this is, from the mappers to the reducers.

The approach would dramatically improve memory throughput as long as the data is localized to specific processors. On the downside, it is expensive to move data from one processor to another, as in workload balancing. Therefore, the solution is attractive for workloads where the same sub-function is used for different data sub-sets or computations composed by several sub-functions interested on different data chunks and few dependencies, a typical MapReduce computation.

**Large rings composed by SMP nodes.**

The 64 virtual CPUs could be seen as normal DaCy nodes. With 16 nodes from

\(^3\)http://research.microsoft.com/en-us/projects/daytona/
Huygens\textsuperscript{4}, a ring could be composed by 1024 virtual nodes. Furthermore, with Infini-Band DDR on PCI Express version 2.0, network loopback has a performance near to a shared message passing implementation. Authors in [98] have shown that for Infini-Band QDR it outperforms shared memory message passing implementation.

The large ring would be organized in two rings, the inner-ring and outer-ring (cf., Section 6.5.3 and Figure 6.12).

"A inner-ring is simplified version of a chordal ring, a simple ring with cross or chordal links between nodes on opposite sides. With different dimensions a chordal ring is used to define several virtual rings to speed up data forwarding of a single hot-set in a large ring."

In this case, the inner-ring is the physical connection between the 16 peers. The outer-ring is defined by the 1024 nodes. Using loop-back, scheduling data movements between the node within the same peer is trivial. Through the requests catalog a local node is identified and a data chunk can be placed directly in its memory space with a single hop. Furthermore, the first peer’s node is used as the connector to the inner-ring. If a data chunk is not required by any of the peer’s nodes, the data chunk is automatically forwarded to the next peer. It reduces data access latency by avoiding the hop of 63 nodes before it reaches the next peer.

With this soft virtualization model, DaCyDB is able to operate with 1024 simple virtual nodes, connected through an efficient Data Cyclotron layer, and achieve high performance. Furthermore, the study of this approach would open the ground for an integration of the Data Cyclotron architecture with a MapReduce framework for efficient data access and data distribution between map and reduce phases, or even independent MapReduce jobs such as in Pig.

\textbf{Blue Gene.}

Blue Gene is another type of computer which could raise interest since they are known for their computational power. Blue Gene is an IBM project aimed at designing supercomputers that can reach operating speeds in the PFLOPS (petaFLOPS) range, with low power consumption. The major characteristic of these computers is the interconnection of nodes, a five-dimensional torus interconnect the compute nodes in a peer-2-peer mode. However, the architecture of a Blue Gene, such as Blue Gene/Q, is not suited for the Data Cyclotron application scenarios.

A Blue Gene/Q cluster is composed of compute nodes and storage nodes. Each

\textsuperscript{4}To distinguish them from the virtual nodes from now on they are referred as peers.
compute chip gets contains 10 links with 2 GB/sec each (they assure 4 GB/sec bi-directional bandwidth), plus one extra 2 GB/sec link to with I/O nodes. This means each compute node is able to receive and send 20 GB of data per second. The bandwidth is attractive, however, each node only has 16 GB of main memory. It is an architecture balanced for intense computational applications and with low memory footprint such as floating point algorithms.

7.2.3 Row-store integration

The Data Cyclotron integration has a modest impact on existing query execution engines and distributed applications. The integration is restricted to data type independent exchange and three calls to define which data is needed, when it is needed, and when it is released. The DaCyDB prototype showed how easy it was to integrate the Data Cyclotron software with a column-store. To consummate such a fact we here describe the integration with a typical row-store DBMS.

A generic row store DBMS is composed of many components, but it is at the back-end where iteration over the data takes place. In these systems, the usual query lifecycle involves the following stages: the parser, rewriter, planner, and executor. The parser creates a parse tree using these object definitions and passes it to the rewriter. The rewriter retrieves any rules specific to database objects accessed by the query, rewrites the parse tree using these rules, and passes the rewritten parse tree to the planner. The planner, or the optimizer, finds the most optimal path for the query execution by looking at collected statistics or some other metadata. An execution plan is then passed to the executor. The main function of the executor is to fetch data needed by the query and pass the query results to the client.

An execution plan is composed of several operators such as scan methods, join methods, aggregation operations, etc. Generally, the operators are arranged in a tree as illustrated in Figure 7.1. The execution starts at the root of the tree and moves down to the next level of operators in case the input is not available. Hence, the data flows from the leaves towards the root and the output of the root node is the query result.

The first rows for processing, i.e., the raw-data, are fetched at the bottom of the tree by the scan operators. A scan operator receives the identification of the table and starts reading data from a storage device. The rows are retrieved one at the time or in batches.

Integration.

The planner will be the one responsible to insert calls into the query plan based on the catalog built during the data distribution among the nodes. To keep the interaction with the Data Cyclotron transparent for all operators, the most reasonable place for the
"umbilical cord" between the DBMS and the Data Cyclotron is at the root and the bottom of the tree.

The requests will be injected at the root of a tree to inform the Data Cyclotron upfront about all input tables. For long plans these injections are pushed down into the root of sub-trees to avoid data floods in the ring.

Due to the nature of the scan operator, the `pin()` and `unpin()` calls will be embedded into a single operator called `hold()` which would replace the scan operator in the plan. The `hold()` pins the table once the first row is requested and `unpins` once the last row is fetched.

The interaction between the row-store DBMS and the Data Cyclotron will have the cooperative work between the queries as described for MonetDB. The intra-query parallelism will also be explored using data partitioning.

**Data partitioning.**

Access to a full table is not feasible within a single buffer. Hence, a main table has to be sliced into sub-tables where each of them fits into a DaCy buffer. As for MonetDB, the tables will be sliced horizontally and then distributed uniformly among all nodes.

Depending on the application, horizontal partitioning might not be enough to define
an optimal hot-set. For a workload with queries using a few columns from each table, the data-chunks would carry unused data, wasting bandwidth and memory bus. Vertical partitioning, as a column-store, emerges as the right solution, however, to still explore the features and advantages of row-stores, the number of columns on each partition should then be defined based on the workload. Similar to RCFiles used in MapReduce, the tables would then be sliced horizontally based on value ranges, but each partition would then be vertically sliced into sub-groups of columns.

The ideal scenario would be a partition scheme, that, based on the workload, re-adapts the partitions over time. For example, creating vertical sub-partitions out of existing partitions, join sub-partitions into a single partition, or move columns from one partition to another. The discussion of this optimization is under research as well as the integration of a row-store, such as PostGres, with the Data Cyclotron.

7.3 Looking back, to look forward

The Data Cyclotron (DaCy) allied with novel and pioneer column store technology created DaCyDB. It addresses the grand challenge for distributed data processing: a self-organizing architecture which exploits all hardware resources for the current workload, achieves an accurate database subset definition, minimizes response time, and maximizes throughput without a single point for global co-ordination.

With an emphasis on simplicity and the autonomous behavior of each component, old concepts were revived, controversial methods were adopted, and orthogonal ideas were combined in novel ways. The presented, discussed, and fully functional prototyped architecture is the result of a thoroughly research exercise to define a new research direction for distributed data processing. It confirms our conviction that this type of challenging research is the right approach to reach new optimums in the database world.