# The Data Cyclotron Query Processing Scheme

ROMULO GONCALVES
MARTIN KERSTEN
CWI Amsterdam

A grand challenge of distributed query processing is to device a self-organizing architecture, which exploits all hardware resources optimally to manage the database hot set, minimize query response time, and maximize throughput without single point global co-ordination. The Data Cyclotron architecture[Goncalves and Kersten 2010] addresses this challenge using turbulent data movement through a storage ring built from distributed main memory and capitalizing on the functionality offered by modern remote-DMA network facilities. Queries assigned to individual nodes interact with the storage ring by picking up data fragments, which are continuously flowing around, i.e., the hot-set.

The storage ring is steered by the *level of interest* ($LOI$) attached to each data fragment, which represents the cumulative query interest as it passes around the ring multiple times. A fragment with $LOI$ below a given threshold, inversely proportional to the ring load, is pulled out to free up resources. This threshold is dynamically adjusted in a fully distributed manner based on ring characteristics and locally observed query behavior. It optimizes resource utilization by keeping the average data access latency low. The approach is illustrated using an extensive and validated simulation study. The results underpin the fragment hot-set management robustness in turbulent workload scenarios.

A fully functional prototype of the proposed architecture has been implemented using modest extensions to MonetDB and runs within a multi-rack cluster equipped with Infiniband. Extensive experimentation using both micro benchmarks and high-volume workloads based on TPC-H demonstrates its feasibility.

The Data Cyclotron architecture and experiments open a new vista for modern distributed database architectures with a plethora of new research challenges.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*Distributed Databases*

General Terms: Design, Performance, Management

Additional Key Words and Phrases: Throughput, Infiniband, Column-stores

## 1. INTRODUCTION

The motivation for distributed query processing has always been to efficiently exploit a large resource pool: $n$ nodes can potentially handle a larger workload more efficiently than a single node. The challenge is to make the queries meet the required data somewhere in the network, taking into account various parameters crucial in a distributed setting, e.g., network latency, load balancing, etc.

The state of the art has evolved from static schemes over a limited number of processing nodes to architectures with no single point of failure, high resistance to network churn, flexible replication policies, efficient routing protocols, etc. [DeWitt and Gray 1992; Kossmann 2000; Yu and Chang 1984]. Their architectural design is focused on two orthogonal, yet intertwined issues: data allocation and workload behavior.

In most architectures a node is made a priori responsible for a specific part of the database using, e.g., a key range or hash function. The query optimizer exploits the allocation function by contracting subqueries to specific nodes or issuing selective data movements and data replication between nodes. Unfortunately, the optimizer search space increases significantly too, making it harder to

Authors' address: Centrum Wiskunde & Informatica (CWI), Science Park 123, 1098 XG Amsterdam, The Netherlands; email:{r.a.goncalves,martin.kersten}@cwi.nl.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

exploit resources optimally. This static data allocation calls for a predictable workload to device optimal system configurations.

Workload behavior is, however, often not predictable. Therefore, an active query monitoring system is needed, e.g., a database design wizard [Chaudhuri and Narasayya 2007; Zilio et al. 2004; Lee et al. 2000] to advice on indices and materialized views, and followed up with scheduled database maintenance actions. The problem is magnified in the presence of a skewed query workload in combination with a poor data allocation function.

Furthermore, workload characteristics are not stable over time either, i.e., datawarehouses and scientific database applications shift their focus almost with every session. In a datawarehouse, analysts shift their focus on a daily basis to address new business opportunities, e.g., a change in market behavior, or the weather may call for a different data set to be explored. In such scenarios, the workload continuously shifts from one part of the network to another making it extremely hard to find optimal data allocation scenarios given the high cost to reshuffle data kept on disks.

In a scientific database there is a continuous battle for scarce main memory and processing resources. Their analytical applications are like a "magnifying glass" searching for the unknown in huge data-sets. The search moves from one database sub-set to another one, zooming in and out until the complete data space has been explored. For example, an astronomer searching for a new star in a space catalog, e.g., the SkyServer catalog [Szalay et al. 2002; Gray et al. 2002; Ivanova et al. 2007].

Both business and science datawarehouse cases lead to a short retention period for data- and workload- allocation decisions which may cause the resource utilization to deteriorate. Those applications require a generic and simple data-access as well as the flexibility to change the search focus, allocate/deallocate resources and high throughput with modest latency.

Hence, the ultimate goal is to design a self-organizing architecture, which maximizes resource utilization without global coordination, even in the presence of skewed and volatile workloads [Chaudhuri and Narasayya 2007]. There is certainly room to design new data allocation functions [Hababeh et al. 2007], grid-base algorithms [Bose et al. 2007], distributed optimization techniques and associated workload scheduling policies [Märtens et al. 2003]. Also several companies, e.g., Greenplum, Asterdata, Infobright, exploit the cluster and compute cloud infrastructures to increase the performance for business intelligence applications using modestly changed commodity open-source database systems. Even reduced, but scalable database functionality is entering the market, e.g., SimpleDB (Amazon) and Bigtable (Google).

However, it is clear that following a research track explored for three decades gets us only up to a certain point regarding improved resource utilization. We end up in a local optimum dictated by software inertia to follow the prime hardware trends, such as increased CPU performance and disk bandwidth. It obscures the view of possible better approaches. If such a solution exists.

The hardware trends are on our side, which makes massive processing, huge main memories, and fast interconnects affordable for experimentation with novel architectures [Govindaraju et al. 2006]. Therefore, in the spirit of an earlier attempt in this field [Herman et al. 1987; Banerjee and Li 1994], the *Data Cyclotron* architecture [Goncalves and Kersten 2010] was designed. It is a different approach to distributed query processing by reconsidering de-facto guidelines of the past [Kersten 2008]. It provides an outlook on a research landscape barely explored.

The Data Cyclotron detaches itself from an old premise in distributed query processing, i.e., most distributed database systems from the past concentrate on reducing network cost. This was definitely a valid approach for the prevalent slow network connections. However, the continuous technology advancement calls for reconsideration of this design guideline. Nowadays, we can have network connections of 1.25 GB/s (10 Gb/s Ethernet) and more. Furthermore, Remote Direct Memory Access (RDMA)[1] technology enables fast transfer of complete memory blocks from one machine to another in a single (fast) step, i.e., without intervention of the operating system software stack. Re-

[1] http://www.rdmaconsortium.org/

ceiving data stored in main memory of a remote node, even ignoring the disk latency, can be as fast as (and even faster than) a state-of-the-art RAID system with its typical bandwidth of 200-800 MB/s.

RDMA is widely available in compute clusters, yet it remains a bit unknown in the database community. Its functionality and application, as well as the benefits of using such modern hardware are summarized in section 2.3.

With the outlook of RDMA and fast interconnects, the Data Cyclotron is designed around processing nodes, comprised of state-of-the-art multi-core systems with sizable main memories, RDMA facilities, running an enhanced database system, and a Data Cyclotron service. The Data Cyclotron service organizes the processing nodes into a *virtual storage ring* topology used to send the *database hot set* continuously around[2]. A query can be executed at any node in the ring; all it has to do is to announce its data interest and to wait for it to pass by.

This way, continuous data movement becomes the key architectural characteristic of the Data Cyclotron. It is what database developers have been trying to avoid since the dawn of distributed query processing. It immediately raises numerous concerns and objections, which are better called research challenges. The query response time, query throughput, network latency, network management, routing, etc., are all areas that call for a re-evaluation. We argue that hardware trends call for such a fresh, but unorthodox, look and match it with a thorough assessment of the opportunities they create. Hence, the Data Cyclotron uses continuous data movement as the pivot for a continuous self-organizing system with load balancing.

Adaptation to changes in the workload is an often complex, expensive and slow procedure. With the nodes tightly bound to a given data set, we first need to identify the workload change and the new workload pattern, then assign the new responsibilities, move the proper data to the proper nodes, and let everyone know about the new distribution.

In the Data Cyclotron a workload change affects the hot data set on the ring, which is triggered by query requests for data access. Its self-organization in a distributed manner, keeping an optimal resource utilization, replaces gradually the data in the ring to accommodate the current workload. During this process a high query throughput and a low query latency is assured.

Furthermore, since a node is not assigned to any specific responsibility other than to manage hot data in its memory buffers and cold data on its attached disks, the queries are not tied to be executed to any specific node or group of nodes. Instead, each query searches a lightly loaded node to execute on; the data needed will pass by. 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.

These design characteristics affect the query optimizers significantly. Less information is available to select an optimal plan, i.e., the whereabouts of data are not known a priori, nor is there control over the storage ring infrastructure for movements of intermediate results. Instead, the optimal processing of a query load is a collective responsibility of all nodes, reflected in the amount and the frequency of data fragments flowing through the ring. The effect is a much more limited role for distributed query optimizers, cost models, and statistics, because decisions are delegated to the self-organizing behavior of the Data Cyclotron service components that maintain the hot data set flow, the data fragments frequency on the storage ring, and possible replicas flowing around.

The remainder of this paper is organized as follows. Section 2 provides a short introduction to the database engine used in our prototype and the state of the art in networking. Section 3 describes in detail the Data Cyclotron architecture and algorithms, Section 4 presents the hot-set management for the Data Cyclotron and provides an in depth analysis through simulation. Section 5 shows the harmonious integration of the Data Cyclotron with hardware and applications. Section 6 presents the prototype and validates the architecture and its protocols with a well-known benchmark, TPC-H. An overview of the opportunities created by the architecture that call for more intense research is

[2]For convenience and without loss of generality, we assume that all data flows in the same direction, say *clockwise*.

given in Section 7. The positioning of our work is covered in the related research Section 8 followed by a wrap up of the main ideas and contributions of this work in Section 9.

## 2. BACKGROUND

The Data Cyclotron is designed from the outset to extend the functionality of an existing DBMS. The DBMS chosen is MonetDB, since its inner workings are well known and its developers provided guidance on the internals of the system. Its basic building blocks, its architecture, and its execution model are briefly reviewed using a SQL:2003 example [3].

The key feature of the Data Cyclotron is its continuous data flow between remote memories using a ring topology. The design was motivated by the road map for network hardware, Infiniband and its RDMA technology. We review the state of the art for improved frame of reference.

### 2.1. DBMS Architecture

MonetDB is a modern fully functional column-store database system [Manegold et al. 2009; Boncz et al. 2008; Boncz et al. 2002]. It stores data column-wise in binary structures, called Binary Association Tables, or BATs, which represent a mapping from an OID to a base type value. The storage structure is equivalent to large, memory-mapped, dense arrays. It is complemented with hash-structures for fast look up on OID and attribute values. Additional BAT properties are used to steer selection of more efficient algorithms, e.g., sorted columns lead to sort-merge join operations.

The software stack of MonetDB consists of three layers. The bottom layer, the kernel, is formed by a library that implements a binary-column storage engine. This engine is programmed using the MonetDB Assembly Language (MAL). The next layer, between the kernel and front-end, is formed by a series of targeted query optimizers. They perform plan transformations, i.e., take a MAL program and transform it into an improved one. The top layer consists of front-end compilers (SQL, XQuery), that translate high-level queries into MAL plans.

The SQL front-end is used to exemplify how a MAL plan is created. An SQL query is translated into a parametrized representation, called a query template, by factoring out its literal constants. This means that a query execution plan in MonetDB is not optimal in terms of a cost-model, because range selectivity do not have a strong influence on the plan structure. They do, however, exploit both well-known heuristic rewrite rules, e.g., selection push-down, and foreign-key properties, i.e., join indices. The query templates are kept in a query cache.

```
select c.t_id from t, c
where c.t_id = t.id;
```

From top to bottom, the query plan localizes the persistent BATs in the SQL catalog for ID and T_ID attributes using the bind operation. The major part is the binary relational algebra plan itself. It contains several instruction threads, starting at binding a persistent column, reducing it using a filter expression or joining it with another column [4], until the results tuples are constructed. The last part constructs the query result table.

The query template is processed by an SQL-specific chain of optimizers before taking it into execution. The MAL program is interpreted in a dataflow driven fashion. The overhead of the interpreter is kept low, well below one $\mu$sec per instruction.

```
begin query();
  X1 := bind("t","id");
  X6 := bind("c","t_id");
  X9 := reverse(X6);
  X10 := join(X1, X9);
  X14 := reverse(X10);
  X15 := leftjoin(X14, X1);
  X16 := resultSet(X15);
  X22 := io.stdout();
  exportResult(X22,X16);
end query;
```

Fig. 1: Selection over two tables

[3] The system including our extensions can be downloaded from http://www.monetdb.org

[4] The reverse is an internal operator to invert the BAT for a join on the OIDs.

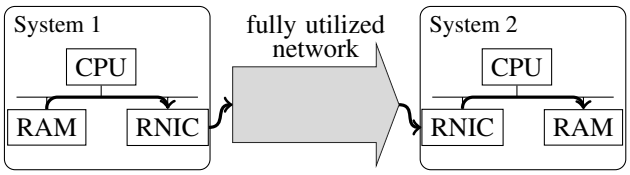



Fig. 2: Network transfer using RDMA. RNICs handle data transfer autonomously; data has to cross each memory bus only once[Frey et al. 2010].

### 2.2. Infiniband Technology

The Data Cyclotron is designed for both Ethernet and Infiniband. However, Infiniband is preferred for its better technology in latency and bandwidth management. Moreover, it does not put limitations on the logical topologies to be implemented and it makes efficient use of the underlying physical topology.

Infiniband is a multi-lane protocol. Generally speaking, the four-lane (4x) products are used to link servers to switches, the eight-lane (8x) products are used for switch uplinks, and the twelve-lane (12x) products are used for switch-to-switch links. The single-lane (1x) products are intended to run the Infiniband protocol over wide area networks.

In 2000, the original Infiniband ran each lane at 2.5Gb/sec (single data rate). However, in 2005, the double data rate (DDR) pushed it up to 5Gb/sec and since 2008, the QDR (quad data rate) products ran each lane at 10Gb/sec. Starting in 2011, there will be two new lane speeds for Infiniband using 64/66 encoding. The first is FDR (fourteen data rate) with 14Gb/sec speed. The second is called EDR, short for Eight Data Rate, which actually runs the Infiniband lanes at 25Gb/sec. A bandwidth of 300 Gb/s bandwidth between any two nodes using 12x EDR comes within reach [5]. With this road map, the bottleneck in data exchange is shifting to the PCI Express bus. The PCI bus technology is still based on old encoding techniques and can not catch up with the Infiniband bandwidth evolution. However, Infiniband has a technology to reduce the memory BUS overhead in data transfer. This technology is called Remote Direct Memory Access (RDMA) technology and it is presented in detail below.

### 2.3. Remote Direct Memory Access

RDMA enables direct access to the main memory of a remote host (read and write). While it allows a simple network protocol design, it also significantly reduces the local I/O cost in terms of memory bus utilization and CPU load when transferring data at very high speeds.

*2.3.1. Applying RDMA.* Remote Direct Memory Access (RDMA) moves data from the memory of one host to the memory of another host by a specialized network interface card called *RDMA-enabled NIC (RNIC)*.

Before starting network transfers, application memory regions must be explicitly registered with the network adapter. This serves two purposes: first, the memory is pinned and prevented from being swapped out to disk; secondly, the network adapter stores the physical address corresponding to the application virtual address. Now it is able to directly access local memory using its DMA engine and to do remote data exchanges. Inbound/outbound data can directly be placed/fetched by the adapter to/from the address in main memory supplied by the application.

Figure 2 illustrates a typical RDMA data path: thanks to the placement information, the RNIC of the sending host can fetch the data directly out of local main-memory using DMA. It then transmits the data across the network to the remote host where a receiving RNIC places the data straight in its destination memory location. On both hosts, the CPUs only need to perform control functionality,

[5] http://www.infinibandta.org/content/pages.php?pg=technology_overview

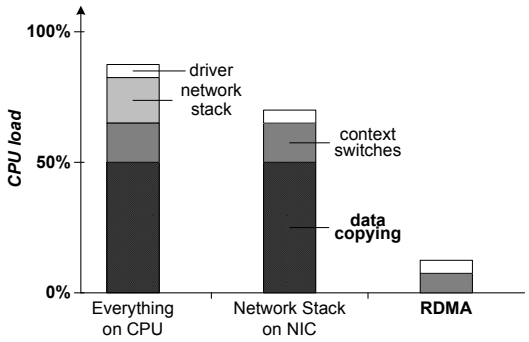



Fig. 3: Only RDMA is able to significantly reduce the local I/O overhead induced at high speed data transfers[Frey et al. 2010].

if at all. Data transfer is done entirely by the RNICs. They can handle high-speed network I/O ($\geq$ 10 Gb/s) between two hosts with minimal involvement of either CPU.

*2.3.2. RDMA Benefits.* The most apparent benefit of using RDMA is the CPU load reduction thanks to the aforementioned direct data placement (avoid intermediate data copies) and OS bypassing techniques (reduced context switch rate) [Balaji 2004]. A rule of thumb in network processing states that about 1 GHz in CPU performance is necessary for every 1 Gb/s network throughput [Foong et al. 2003]. Experiments on our cluster confirmed this rule: even under full CPU load, our 2.33 GHz quad-core system was barely able to saturate the 10 Gb/s link.

Figure 3 depicts the CPU load breakdown for legacy network interfaces under heavy load and contrasts them with the latest RDMA technology. As can be seen, the dominating cost is the intermediate data copying—required by most legacy transport protocols—to transfer data between the network and the local main memory. Therefore, offloading only the network stack processing to the NIC is not sufficient (middle chart), but data copying must be avoided as well. Only RDMA is currently able to deliver such a high throughput at negligible CPU load.

A second effect is less obvious: RDMA also significantly reduces the memory bus load as the data is directly DMAed to/from its location in main-memory. Therefore, the data crosses the memory bus only once per transfer. The kernel TCP/IP stack on the other hand requires several such crossings. This may lead to noticeable *contention* on the memory bus under high network I/O. Thus, adding additional CPU cores to the system is *not* a replacement for RDMA.

*2.3.3. Perfect Match.* By design, the RDMA interface is quite different from a classical Socket interface. A key difference is the asynchronous execution of the data transfer operations which allows overlapping of communication and computation, thereby hiding the network delay.

Taking full advantage of RDMA is not trivial as it has hidden costs [Frey and Alonso 2009] with regard to its explicit buffer management. The ideal scenario is when the buffer elements are large, preferably a few dozen up to hundreds of megabytes. Furthermore, to alleviate expensive memory registration, the node interconnections topology should be point to point [Frey et al. 2010].

Due to these costs and functional requirements, not every application can fully benefit from RDMA. However, a database architecture like the *Data Cyclotron* clearly can. It aims to transfer big data fragments using a ring topology, i.e., each node has a point to point connection with its neighbors.

*2.3.4. A Burst for Distributed Processing.* Through the years, the data access latency for DBMS has been improved. Different algorithms or structures were created to exploit the hardware improvements for an efficient data access. RDMA, an advancement in network hardware, is a recent candidate to explore new methods for efficient data access.

The RDMA made possible to access to remote memory as if it was to a local storage. Due to the DMA facility, the data processing can overlap the data access. Furthermore, the data can be fetched to the local memory data at higher speeds due to the bandwidth provided by Inifiniband.

The Data Cyclotron [Goncalves and Kersten 2010] and *cylo-join* [Frey et al. 2010; 2009] are on the front line to capitalize those features. They use the RDMA buffers, i.e., the registered memory regions, as a *drop-box* to shuffle data between nodes removing the dependency from slow disks. They explore the effortless and efficient data movement to develop new ideas for distributed data access and design of distributed relation operators.

In Data Cyclotron, the data structures are placed into the remote memory without effort and keeping the process transparent for the DBMS. In [Frey et al. 2010] for $(R \bowtie S)$, one relation, say S (partitioned into sub-relations $Si$) is kept stationary during processing while the fragments of the other relation, say R, are rotating. Hence, all ring members join each fragment of $R$ flowing by against their local piece of $S$ ($S_i$) *locally* using a commodity in-memory join algorithm.

## 3. DATA CYCLOTRON ARCHITECTURE

The Data Cyclotron (DaCy) system architecture is initially built around a ring with homogeneous nodes. Each node consists of three layers: the DBMS layer, the Data Cyclotron layer, and the network layer (see Figure 5).

The Data Cyclotron layer contains the DaCy runtime itself and a DaCy data loader. The network layer encapsulates the envisioned RDMA infrastructure and traditional UDP/TCP functionality as a fall-back solution.

The system is started with a list of nodes to participate in the ring. The data is spread upfront over the disks attached to the nodes using any feasible partitioning scheme. Details on the data partition and distribution are presented later in Section 5.2. For the remainder, we assume each partition to be an individual BAT comfortably fitting the main memory of the individual nodes. Furthermore, these BATs are randomly assigned to ring nodes where the local DaCy data loader becomes their owner and administers them in its own catalogs (Structure S1 Figure 5). The BAT owner node is responsible for putting it into or pulling it out of the hot set occupying the storage ring.

```
begin query();
  X1 := bind("t","id");
  X6 := bind("c","t_id");

  X9 := reverse(X6);


  X10 := join(X1, X9);
  X14 := reverse(X10);
  X15 := leftjoin(X14, X1);

  X16 := resultSet(X15);
  X22 := io.stdout();
  exportResult(X22,X16);
end query;
```

(a) MAL structure

```
begin query();
  X2 := request("t","id");
  X3 := request("c","t_id");
  X6 := pin(X3);
  X9 := reverse(X6);
  unpin(X3);
  X1 := pin(X2);
  X10 := join(X1, X9);
  X14 := reverse(X10);
  X15 := leftjoin(X14, X1);
  unpin(X2);
  X16 := resultSet(X15);
  X22 := io.stdout();
  exportResult(X22,X16);
end query;
```

(b) After the DaCy Optimizer

Fig. 4: Selection over two tables

Infrequently used BATs are retained on a local disk at the discretion of the DaCy data loader. The detailed behavior of each layer and their interaction is studied in more detail below.

### 3.1. The DBMS Layer

The MonetDB server receives an SQL query and compiles it into a MAL plan. This plan is analyzed by the Data Cyclotron optimizer, which injects three calls *request()*, *pin()* and *unpin()*. They exchange resource management information between the DBMS layer and the DaCy runtime.

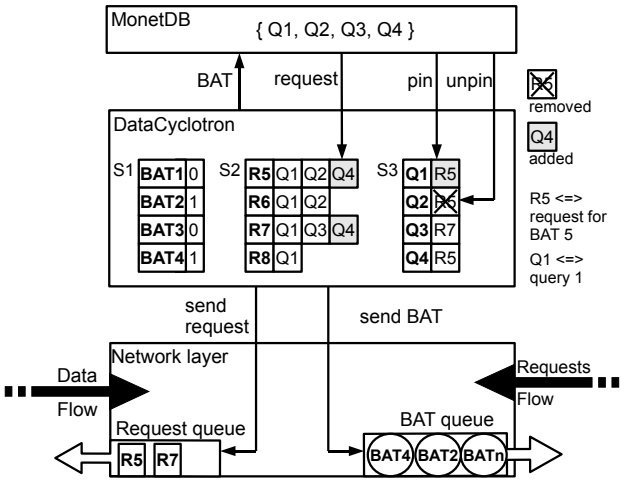



Fig. 5: Internal node structure

More precisely, the DaCy optimizer replaces each *bind()* call into the SQL catalog by a *request()* call to the ring and keeps a list of all outstanding BATs requests. For each relational operator, it checks if any of its arguments should come from the Data Cyclotron layer. The first occurrence leads to injection of a *pin()* call into the plan. Likewise, the last reference is localized and complemented with an *unpin*() call.

Figure 4b illustrates a sample MAL program from Figure 4a after being massaged by the DaCy optimizer. This MAL plan is executed using a multithreaded interpreter following the dataflow dependencies. The *request()* and *unpin()* calls are non-blocking operators, the *pin()* call blocks until the BAT has become available at the DBMS layer.

As a side note, observe that this optimization scheme can be readily applied in traditional row-store database engines by injection of these operators in conjunction with scan operators. (See Section 8.3).

### 3.2. The Data Cyclotron Layer

The Data Cyclotron layer is the control center and it serves three message streams, those composed by a) the requests from the local MonetDB instance, b) the predecessor's BATs, and c) the successor's requests from the network layer. The catalog structure *S1* contains information about all BATs owned by the local node. The structure *S2* administers the outstanding requests for all active queries, organized by BAT identifier. The structure *S3* contains the identity of the BATs needed urgently as indicated by the *pin* calls.

*3.2.1. DBMS and Data Cyclotron interaction.* Consider the steps taken to execute the plan in Figure 4b. If the BAT is owned by the local DaCy data loader, it is retrieved from disk or local memory and put into the DBMS space. Otherwise, the call leads to an update of *S2* (the shaded squares in Figure 5). A BAT request message is then sent off to the ring, traveling anti-clockwise towards its owner. Thereafter, the BAT travels clockwise towards the requesting node. The *pin()* request checks the local cache for availability. If it not available, query execution blocks and the *pin()* call is stored in catalog *S3*. It remains blocked until the BAT is received from the predecessor node. The BAT request is removed from *S3* by the *unpin()* call. The interaction between the layers ends with the last *unpin()* call, which removes the query from *S3*.

*3.2.2. Peer interaction.* The *Request Propagation* algorithm, illustrated in Figure 6, handles the requests in *S2*. There are several possible outcomes for this algorithm. If the message is received

*Request Propagation*
***input:*** the request node's origin `owner` and the request id `reqid`
***output:*** forwards the request or schedules the load of the requested BAT

```
01: /* check if the request returned to its origin */
02: if ( owner == node_id )
03:    unregister_request( &S2, reqid );
04:    unregister_request_queries ( &S3, reqid );
05:    exit;
06:
07: /* check if the node is the BAT owner */
08: if ( node_is_owner_( &S1, reqid ) )
09:    if ( data_chunk_is_loaded( &S1, reqid) )
10:        exit;
11:    if ( data_chunk_can_be_loaded( reqid ) )
12:       if ( data_chunk_is_already_pending(reqid) )
13:          data_chunk_load( reqid, chk_size);
14:       untag_data_chunk_pending(reqid);
15:       exit ;
16:    else
17:        if ( !data_chunk_is_already_pending(reqid) )
18:           tag_data_chunk_pending(reqid);
19:           exit;
20:
21: /* check if there is the same request locally */
22: if ( request_is_mine(reqid) )
23:    if ( !request_is_sent(reqid) )
24:       /* send if it has not been sent */
25:       load_request(node_id, reqid);
26:       exit;
27:
28: forward_request(owner, reqid);
```

Fig. 6: Request Propagation Algorithm

by the *request originator* node, i.e., the BAT request is back to its origin, it is unregistered from *S2* and the associated queries raise an exception; the BAT does not exist (anymore) in the database. Alternatively, if the node is the BAT owner, but the BAT was already (re-) loaded into the hot set, the request can be ignored. If the BAT was not yet loaded, but the storage ring is full it is marked as *pending*, i.e., the load is postponed until hot set adjustment decisions are made. If the ring is not full, the BAT is loaded from the node's local storage and becomes part of the storage ring hot set. In the case we receive a duplicate for an outstanding request, the request is absorbed. Otherwise, it is just forwarded.

The final outcome of a request is to make a BAT part of the hot set, i.e., to enter the storage ring, and travel from node to node until it is not needed anymore, thereby removed by its owner. BAT propagation from the predecessor node to the successor node is carried out by the *Data Chunk Propagation* algorithm as depicted in Figure 7.

For each BAT received, the algorithm searches for an outstanding request in *S2*. Once found, it checks the catalog *S3* and unblocks related queries blocked in a *pin()* call handing over the BAT received as a pointer to a memory mapped region, i.e., the buffer where it is located. For example, in Figure 4a, a reference for the BAT *t_id*, is passed through the variable *X1*. The buffer is freed, i.e., made available to received more BATs, by the *unpin()* call.

The Data Cyclotron data loader is aware of the memory consumption in the local node only. If there is not enough buffer space, the BAT will continue its journey and the queries waiting for it remain blocked for one more cycle.

*3.2.3. Storage Ring Management.* The BATs in the storage ring carry an administrative header used by its owner for hot set management. The *Data Chunk Propagation* algorithm (Fig. 7) updates the resource variables *hops* and *copies*. The former denotes the number of hops since it left its

*Data Chunk Propagation*
***input:*** the BAT loader's id `owner`, `data_chunk_id`, `loi`, `copies`, `hops` ,`cycles`
***output:*** forwards the BAT

```
01: /* check if there is a local request for the BAT */
02: hops++;
03: if ( data_chunk_has_request(data_chunk_id) )
04:    request_set_sent(data_chunk_id);
05:
06:    if ( request_has_pin_calls( data_chunk_id ) )
07:       copies++;
08:       /*check if it was pinned for all the associated queries */
09:       if ( request_is_pinned_all( data_chunk_id ) )
10:          request_unregister( data_chunk_id);
11:
12: forward_data_chunk(owner, data_chunk_id, loi, copies, hops, cycles);
```

Fig. 7: Data Chunk Propagation Algorithm

owner, a metric for the age of the BAT on the storage ring. The variable *copies* counter designates how many nodes actually used it for query processing.

The runtime system has two more functions for resources management. A *resend()* function is triggered by a timeout on the rotational delay for BATs requested into the storage ring. It indicates a package loss. The *loadAll()* executes postponed BAT loads, i.e., BATs marked as pending in the third outcome of the *Request Propagation* algorithm. Every T msec, it starts the load for the oldest ones. If a BAT does not fit in the *Data Chunk queue*, it tries the next one and so on until it fills up the queue. The leftovers stay for the next call. This type of load optimizes the queue utilization. The priority for entering the storage ring is derived from both the size and the waiting time. These functions make the Data Cyclotron robust against request losses and starvation due to scheduling anomalies.

### 3.3. The Network Layer

The network layer of the Data Cyclotron manages the stream of BATs as data chunk messages. They contain the properties *owner*, *data_chunk_id*, *data_chunk_size*, *loi*, *copies*, *hops*, and *cycles*. If the local node is the BAT's owner, the algorithm *hot data management* (cf., Figure 8) is called for hot set adjustments. Otherwise, it calls the *Data Chunk Propagation* algorithm (Fig. 7). Data Chunk request messages contain the variables, *request owner* and *data_chunk_id*. The *Request Propagation* algorithm (Fig. 6) is called for this type of message. All messages are managed by the network layer on a first-come-first-serve basis.

The underlying network is configured as asynchronous channels with guaranteed order of arrival. The data transfer and the queues management are optimized depending on the protocol being used. The ring latency and the exploitation of its storage capacity is dependent on success of this optimization. Further details of this interaction between the DaCy and the network is studied in more detail in Section 5.1

## 4. HOT SET MANAGEMENT

The data chunks in circulation are considered hot data. They flow as long as they are being considered important for the query workload. The metric for this is called the level of interest, $LOI$ for short, which fluctuates over time.

The number of *copies*, the number of *hops*, the number of *cycles*, and the previous $LOI$ are used to derive a new level of interest each time it passes at the owner node. The variable *copies* and *hops* are updated at each node. The variable *cycles* is only updated by the data chunk's owner when it completes a cycle. Subsequently, the new *loi* is then calculated as follows:

*New level of interest*
***input:*** the `data_chunk_id`, `loi`, `copies`, `hops`, `cycles`
***output:*** forwards the data chunk or unloads the data chunk.

```
01: /* Check if the node is the data chunk loader */
02: if ( node_is_the_loader(data_chunk_id) )
03:    cycles++;
04:    new_loi =
        (loi + ((copies / hops) * cycles)) /
        cycles;
05:    copies = 0;
06:    hops = 0;
07:    if ( new_loi < loit(n) )
08:       unload_data_chunk( data_chunk_id );
09: else
10:    forward_data_chunk(data_chunk_id, new_loi, copies, hops, cycles);
```

Fig. 8: Hot Data Set Management

$$\begin{aligned} \text{CAVG} &= \frac{\mathit{copies}}{\mathit{hops}} \\ newLOI &= \frac{\mathit{LOI} + \mathit{CAVG} \times \mathit{cycles}}{\mathit{cycles}} = \frac{\mathit{LOI}}{\mathit{cycles}} + \mathit{CAVG} \end{aligned} \tag{1}$$

The previous $LOI$ for a data chunk carries its history of the ring's interest during previous cycles. However, the latest cycle has more weight than the older ones. This weight is imposed by multiplication of the number of copies average $CAVG$ in the last cycle by the actual *cycles* value:

$$\frac{\mathit{copies}}{\mathit{hops}} \times \mathit{cycles} \tag{2}$$

The division by the number of cycles applies an age weight to the formula. Old data chunks carry a low level of interest, unless renewed in each pass through the ring. The new $LOI$ value is then compared with the minimum level of interest maintained per node, i.e., the level of interest threshold $LOIT_n$. If the new $LOI$ is lower than $LOIT_n$ the data chunk is removed from circulation. If not, the $LOI$ variable is set with the new $LOI$ value and the data chunk is sent back to the ring. This *hot data management* algorithm (Figure 8) is executed at the Data Cyclotron layer for all data chunks received from the predecessor node. The minimum level of interest, i.e., $LOIT_n$, is the threshold between what is considered hot data and cold data. Each node has its own $LOIT_n$ and its value is derived from the local *data chunk queue* load.

An overloaded ring increases the probability postponing a data chunk load due to the lack of space. It increases the latency to fulfill a remote request. In this situation, Data Cyclotron reduces the number of data chunks in the ring by increasing the threshold $LOIT_n$. $LOIT_n$ is stepwise increased until the pending local data chunks can start moving. However, if the threshold is increased too much, the life of a data chunk in the ring will reduce. This can lead to a thrashing behavior. The impact of the threshold choice is studied in more detail in 4.1. The model to manage the hot set might not be optimal, but it is robust and behaves as desired.

### 4.1. Simulation

The Data Cyclotron interaction design is based on a discrete event simulation. The core of this activity has been the detailed hot-set study using NS-2 [6], a popular simulator in the scientific community. The simulator runs on a Linux computer equipped with an Intel Core2 Quad CPU at 2.40 GHz, 8 GB RAM and 1 TB disk. The simulator was used out of the box, i.e. without any changes to its kernel.

[6] NS-2 was developed by UC Berkeley and is maintained by USC; cf., http://www.isi.edu/nsnam/ns/

The simulation scenario has a narrow scope, because it is primarily intended to demonstrate and stress-test the behavior of the Data Cyclotron internal policies. Further study is conducted with the fully functional Data Cyclotron on a real-size cluster (See Section 6).

The base topology in our simulation study is a ring composed of ten nodes. Each node is interconnected with its neighbor through a duplex-link with 10 Gb/s bandwidth, 6 $\mu$ delay, and policy based on drop packets from the tail of the queue. Each node contains 200 MB for the *data chunk queue*, i.e., network buffers, which results in a total ring capacity of 2 GB. These network characteristics comply with a cluster from our national super-computer center [7], the target for the fully functional system.

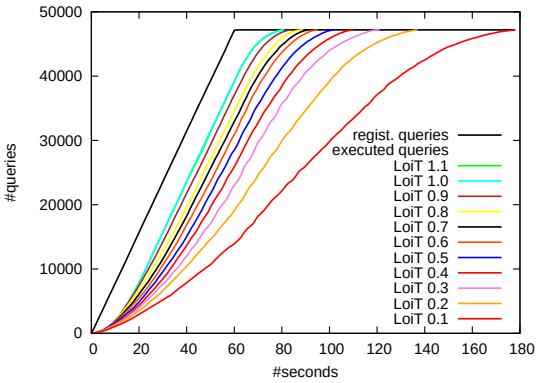
The analysis is based on a raw data-set of 8 GB composed of 1000 BATs with sizes varying from 1 MB to 10 MB. They are uniformly distributed over all nodes, giving ownership responsibility over about 0.8 GB of data per node. The workload is restricted to queries that access remote data only, since we are primarily interested in the adaptive behavior of the ring structure itself.

The first step in the experimentation was to validate correctness of the Data Cyclotron protocols using micro-benchmarks. We discuss three workload scenarios in detail. In the first scenario, we study the impact of the $LOIT_n$ on the query latency and throughput in a ring with limited capacity. In the second scenario, we test the robustness of the Data Cyclotron against skewed workloads with hot sets varying over time.

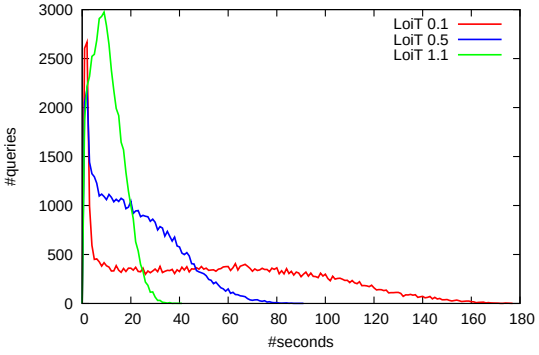
In the third scenario, we demonstrate the Data Cyclotron behavior for non-uniform access patterns.

*4.1.1. Limited Ring Capacity.* The Data Cyclotron keeps the hot-data in rotation by adjusting the minimum level of interest for data chunks on the move. The minimum level of interest ($LOIT_n$) is the threshold which defines if a data chunk is considered hot or cold. A high $LOIT_n$ level means a short life time for them in the ring, and vice versa. The right $LOIT_n$ level and its dynamic adaptation are the issues to be explored with this experiment.

The experiment consists of firing 80 queries per second on each of the 10 nodes over a period of 60 seconds, and then let the system run until all 48000 queries have finished. We use a synthetic workload that consists of queries requesting between one and five randomly chosen remote data chunks. The net query execution times, i.e., assuming all required data is available in local memory, are arbitrarily determined by scoring each accessed data chunk with a randomly chosen processing time between 100 msec and 200 msec.



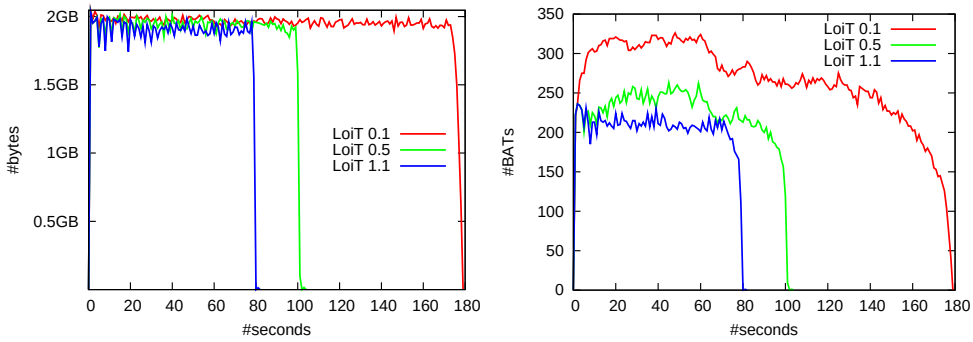

(a) Query Throughput




(b) Query Life time

Fig. 9: Multiple $LOIT_n$ levels

[7] http://www.sara.nl/




(a) Load in Bytes (b) Load in data chunks

Fig. 10: Ring Load

To analyze the impact of $LOIT_n$ on the Data Cyclotron performance behavior, we repeat the experiment 11 times, increasing $LOIT_n$ from 0.1 to 1.1 [8] in steps of 0.1. Between two runs, the ring buffers are cleared, i.e., all the data is unloaded to the local disk.

Figure 9 a) shows the Data Cyclotron throughput for each $LOIT_n$ iteration, i.e., the cumulative number of queries finished over time. The line *registered queries* represents the cumulative number of queries fired to the ring over time.

The experiment shows that a low $LOIT_n$ leads to a higher number of pending queries in the system. For $LOIT_n = 0.1$ at instant 40 seconds, only 8000 out of the 30000 registered queries are finished. However, for $LOIT_n = 1.1$ at the same instant, almost 25000 queries were finished. We observe that the query throughput is monotonously increasing with increasing $LOIT_n$.

Query latency is also affected by low $LOIT_n$ values. The graph in Figure 9 b) shows the query life time distribution (histogram) for three $LOIT_n$ levels. The query life time is its gross execution time, i.e., the time spent from its arrival in the system until it has finished.

The results show that a high $LOIT_n$ leads to lower life time of a query. For example, the $LOIT_n = 0.1$ has a peak in the number of queries resolved in less than 5 secs, but then it has the remaining queries pending for at least 100 seconds.

The reason for these differences stems from the amount of data removed from the ring over the time and the data chunks size. The workload hot set is bigger than the ring capacity which increases the competition for the free space in the ring. Using a low $LOIT_n$, the removal of the hot data chunks is delayed, i.e., the pending loads list at each node grows. Consequently, execution of queries that wait for the pending loads is delayed.

With optimized load for pending data chunks, presented in Section 4, and a low drop rate, the tendency is to leave the big ones for last. Whenever the least interesting data chunk is dropped from the ring, the available slot can only be filled with a pending data chunk of at most the size of the dropped one. Consequently, the ring gets loaded with more and more small data chunks, decreasing the chance of big data chunk loads even further. Only once there are no more pending request for small data chunks, the ring slowly empties, finally making room for the pending bigger data chunks. The graphs in Figure 10 a) and b) identify the data chunk size trend over time. The correlation between the ring load in bytes (Fig. 10a) and the ring load in data chunks (Fig. 10b), shows the data chunk length in the hot set over time. With a continuously overloaded ring and a reduction of the number of data chunks loaded, the graphs depict that the load of big data chunks is being postponed. Therefore, the queries waiting for these data chunks stay pending almost until the end. The delay gets more evident for low $LOIT_n$ levels.

[8] This is the upper limit of the function.

| workload | SW1 | SW2 | SW3 | SW4 |
|---|---|---|---|---|
| skewed | 3 | 5 | 7 | 9 |
| start(sec) | 0 | 15 | 37.5 | 67.5 |
| end(sec) | 30 | 45 | 67.5 | 97.5 |
| queries/sec | 200 | 300 | 400 | 500 |

Table I: Workload details

This experiment confirmed our intuition that the $LOIT_n$ is inversely proportional to the local *data chunk queue* load. It also proofs that $LOIT_n$ should not be static. It should dynamically adapt using the local *data chunk queue* load as a reference. In the next experiment we show how this dynamic $LOIT_n$ behaves when the hot set is changing all the time and how well it exploits the ring resources.

*4.1.2. Skewed Workloads.* The hot-set management is also tested for a volatile scenario. In this experiment we use several skewed workloads *SW*, brute changes in the hot set *H*, and resource competition by the disjoint hot sets *DH*. A skewed workload *SWi* uses a subset of the entire database. The hot set *Hi* used by *SWi* has disjoint data *DHi* not used by any other skewed workload.

Each workload *SWi* can enter the Data Cyclotron at a different times. In some cases they meet in the system, in other cases they initialize after completion of the previous ones. These unpredictable initializations require a dynamic and fast reaction time of the Data Cyclotron. If *SWj* enters the ring while *SWi* is still in execution, the Data Cyclotron needs to arbitrate shared resources between the *DHi* and *DHj*. It must remove *DHi* data chunks with low $LOI$ to make room for the new *DHj* data in order to maintain a high throughput. However, the data chunks from *DHi* to finish *SWi* queries must remain in the ring to ensure a low latency.

This dynamic and quick adaptation should provide answers to three major questions: How fast does the Data Cyclotron react to load data requests for the new workload? Are the queries from the previous workload delayed? How does the Data Cyclotron exploit the available resources?

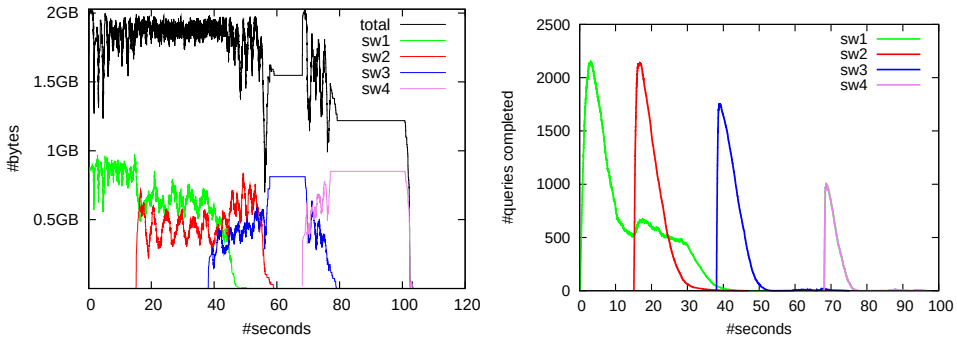
The scenario created has four workloads (*SW1, SW2, SW3, and SW4*). Each *SWi* uniformly accesses a subset of the database (*Di*). Each *Di* has a disjoint subset *DHi*, i.e., *DHi* is not in *Dj, Dk, Dl*, with exception for *DH4* which is contained in *DH1*. Each *Di* is composed by data chunks for which the modulo of their *id* and a *skewed* value is equal to zero. The time overlap percentage between the *SW1* and *SW2* is 50%, 25% for *SW2* and *SW3*, and no overlap for *SW3* and *SW4*. Table I describes each workload.

From the previous experiment, we learned that the $LOIT_n$ should be inversely proportional to the buffer load. The dynamic adaption of the $LOIT_n$ is done using the local buffer load at each node. Every time the buffer load is above 80% of its capacity, the $LOIT_n$ is increased one level. On the other hand, if it drops below 40% of its capacity, the $LOIT_n$ is decreased one level. For this experiment, we used three levels, *0.1, 0.6, 1.1*.

Graph 11 a) shows the space used in the ring by each *DHi*. While, Graph 11 b) shows the amount of queries finished for each *DHi*.

The results illustrate the reactive behavior, i.e. how quickly the Data Cyclotron reacts to the change of workload characteristics. The graph in Figure 11 b) shows a peak of 2000 finished *DH2* queries between the 15th and 16th seconds. The graph 11 a) shows a peak in the load of *DH2* data chunks in the same period. With the initialization of *SW2* at 15 seconds, the peak confirms the quick reaction time. The same phenomenon is visible for all the other workloads.

The ring was loaded with data from *DH2*, however, the data from *DH1* was not completely removed. This is a consequence of the 50% time overlapping between *SW1* and *SW2*. In Graph 11 b) *SW1* queries remain visible until the 43rd second. The data chunks to resolve these queries are kept around as it is shown in Graph 11 a). The Data Cyclotron does not remove all the data from the previous workload in the presence of a new workload until all the queries are finished. It shares




(a) Ring Load (b) Query Throughput

Fig. 11: Skewed workload

the resources between both workloads as predicted. Observe that the sharing of ring resources gets lower as the time overlapping between *SW* decreases.

The *SW3* workload shows an interesting reaction of the Data Cyclotron when it encounters a nearly empty ring. The *DH3* started to be loaded and the ring is near to its limit. The $LOIT_n$ is at its maximum level to free space as much as possible. No more *SW1* and *SW2* queries exist in the system. Therefore, the last data chunks for *DH1* and *DH2* start to be removed from the ring. Their removal brings the ring load down to 37,5% of its capacity, below the 40% barrier defined for this experiment. With this load the $LOIT_n$ is set to its minimum level, i.e, the data chunks are now staying longer in the ring.

With a big percentage of *DH3* queries finished, the Data Cyclotron does not remove the *DH3* data chunks anymore. It keeps loading the missing *DH3* data. The ring gets loaded and it remains with the same load for almost 10 seconds. The Data Cyclotron exploits the available resources by maintaining the *DH3* data chunks longer, i.e, expecting they will be used in the near future.

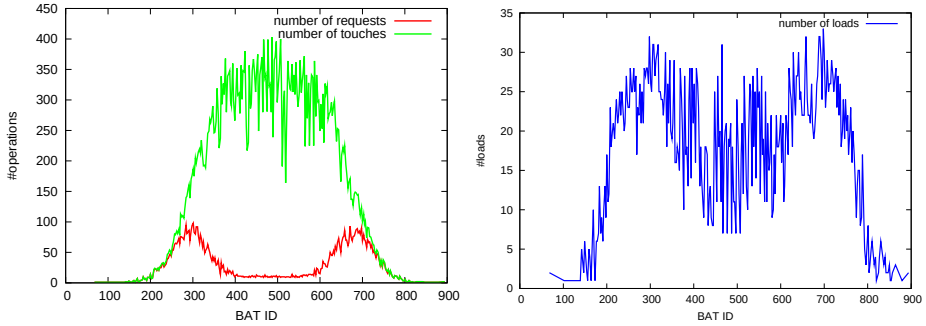
The abundance of resources is over when the *SW4* workload enters the scene. The ring becomes overloaded again, raising the $LOIT_n$ to higher levels. Therefore, the *DH3* data starts being evicted.

*4.1.3. Non-uniform Workloads.* So far we have studied the hot-set management using uniform distributions for the data chunk sizes and keep data access patterns. Leaving the uniform scenarios behind, we move towards workloads with different data access distributions.

In the previous experiments, the study of the data chunk $LOI$ focused on their age. The average number of copies per cycle, i.e., the ring interest, was not included due to the uniform data chunk access patterns. Therefore, we initiated an experiment to stress it using a *Gaussian* data access distribution. The uniform distribution for data chunk sizes is retained. The workload scenario of section 4.1.1 is used with the new data access distribution. The *Gaussian* distribution is centered around data chunk id 500 with a *standard deviation* of 50. All nodes use the same access distribution.

In this kind of workload, the Data Cyclotron keeps the popular data chunks longer in the ring and exploits the remaining ring space for the less popular data chunks. The workload distribution is represented as the green curve in graph 12 a). The *in vogue* group is constituted by the data chunks with id between 350 and 600 which were touched more than 250 times. The data chunks on the edge of this group, the *standard* data chunks, have a lower rate of touches. The remaining ones, with less than 20 touches, are the *unpopular* data chunks.

The *in vogue* data chunks are heavily used by the query workload, i.e., their $LOI$ is always at high levels. Therefore, they are kept in the ring longer. Their low load rate, pictured as blue in graph 12 b), is explained by the Data Cyclotron cold down process. With an overloaded ring and




(a) Copies/Requests Distribution (b) Loads Distribution

Fig. 12: *Gaussian* workload

the $LOIT_n$ at its highest level, the Data Cyclotron removes data chunks to make room for new data. The first ones to be removed are the ones with low $LOI$, i.e., first the *unpopular* then the *standard* data chunks. For this reason the *in vogue* are the ones staying for longer periods as hot data chunks.

The *standard* data chunks are then requested by queries triggering their load. It is this resource management to maintain the latency in low values, that makes the *standard* data chunks to enter and leave the ring more frequently.

The low rate of requests, represented as red, for the *in vogue* data chunks contradicts the common believe that *in vogue* data chunks should be the ones with a high rate of requests, thereby a high rate of loads. The reason stems from the request management in the Data Cyclotron runtime layer. A request is only removed if all its queries *pinned* it. Having a high number of queries entering the system, the probability for an *in vogue* data chunk request to be *pinned* for all its queries at once is too low. As a consequence, the request stays in the node longer.

The experiment results show a good management of the hot set, by the Data Cyclotron, for a *Gaussian* distribution. The high throughput is assured by keeping the *in vogue* data chunks in the ring as long as possible. For a low latency, the $LOIT_n$ is used at its high level to reduce the time to access the many *standard* data chunks.

## 5. TOWARDS A REAL SYSTEM

The Hot-set management was tested through simulation whereas the architecture validation and Data Cyclotron performance comes from a fully functional implementation. We start with a reflection on the symbiotic relationship between the Data Cyclotron and the network hardware as presented in Section 2.2. Subsequently we lay out the features of the Data Cyclotron and MonetDB.

### 5.1. Symbiotic network relationship

The Data Cyclotron architecture has a *"symbiotic relationship"* with the network hardware. Its design leverages the data routing latency and gets optimal bandwidth utilization at networks switches using simplified routing. Furthermore, the Infiniband and RDMA network technology road-map ensures a continuous data flow as a key feature for high throughput.

A network switch is non-blocking if the switching fabric can handle the theoretical maximal load on all ports, such that any routing request to any free output port can be established successfully without interfering other traffics. With non-blocking switches no extra latency is added by the data routing at the switch. Not all software architectures can use and exploit the assets of this type of switches, but the Data Cyclotron can.

In the Data Cyclotron the data flows clockwise, i.e., it is a continuous stream and with a single routing pattern. Furthermore, with the ring topology, the packets that arrived at different input ports are destined to different output ports, i.e., a contention free scheduling. Therefore, they can be

routed instantaneously, i.e, the switches can be non-blocking. Aside from absence of latency, there is ideal switch bandwidth utilization because the routing algorithm is equivalent to synchronized shuffling [Chun 2001].

The Data Cyclotron is designed for applications with high throughput demands. Multiple nodes handle large numbers of concurrent queries. Thanks to Infiniband and RDMA, the Data Cyclotron realizes high speed and low latency data availability at all nodes. In the same line as [Ousterhout et al. 2010], it exploits the fact that remote memory access is becoming more efficient than disk access, i.e., it has lower latency and higher bandwidth.

Typical contemporary storage system architectures, like SATA300 and SAS, provide a theoretical maximum IO performance of 375 MB/s. The effective bandwidth is further degraded by additional hardware overhead, e.g., the seek time, and operating system software time, and especially the random access time.

Remote memory does not suffer from seek and rotational delays. Furthermore, with the sizable Infiniband bandwidth (cf., Section 2.2) and the RDMA benefits (cf., Section 2.3.2), the latency for random data access in remote memories is lower than in the local disk [Ousterhout et al. 2010].

### 5.2. Symbiotic database relationship.

The combination of Data Cyclotron and MonetDB features lead to an effortless data partition/distribution scheme and provision for efficient data access.

Consider the data partitioning and distribution scheme, the Data Cyclotron does not impose any requirement other than that each chunk fits into a single RDMA buffer of several MBs. Hence, the database can be freely partitioned vertically and horizontally using any database partitioning technique to increase locality of access.

The MonetDB optimizer stack contains two modules to perform this task: the *mitosis* and the *mergetable* optimizers. The former takes a query plan and seeks the largest persistent BAT referenced. It will chop it horizontally over the OID list. This is a zero-cost operation, because the underlying storage structure is a BAT view. Simple administration can memory map the portion of interest at any time. The *mergetable* takes the result of the *mitosis* step and unfolds the query plan along all partitions. Each of these sub-plans can be interpreted by a separate thread of control. Blocking operations, such as sort and result set construction, call for assembling the partial results again.

From the Data Cyclotron perspective, the horizontal slices can be seen as column partitions, i.e., data chunks. The queries access each data partition directly from the network buffers. Each partition is a BAT view, whose materialized version fits comfortably in main memory of the individual nodes. With two memory transfers, a DaCy node receives a BAT, makes it available for the DBMS, and forwards it to its left neighbor.

```
begin query();
   X2 := request("t","id",1);
   X3 := request("t","id",2);
   X4 := request("t","id",3);
   X5 := request("c","t_id",1);
   X11 := request("c","t_id",2);
   X12 := request("c","t_id",3);
   X17 := pin(X5);
   X18 := reverse(X17);
   unpin(X5);
   X19 := pin(X11);
   X20 := reverse(X19);
   unpin(X11);
   X21 := pin(X12);
   X22 := reverse(X21);
   unpin(X12);
   X9 := pack(X18,X20,X22);
   X24 := pin(X2);
   X25 := pin(X3);
   X26 := pin(X4);
   X1 := pack(X24,X25,X26);
   unpin(X2);
   unpin(X3);
   unpin(X4);
   X10 := join(X1, X9);          (A)
   X14 := reverse(X10);          (B)
   X15 := leftjoin(X14, X1);     (C)
   X16 := resultSet(1,1,X15);
   X23 := io.stdout();
   exportResult(X23,X16);
end query;
```

Fig. 13: MAL plan after DcOptimizer

Combined with the intra-query parallelization at each node, arrival of a single partition is enough to resume execution of a group of (sub)queries. Furthermore, there is no explicit order to process the column partitions within each individual plan. Instead, the execution method is driven by partition availability, much like the approach proposed for *cooperative scans* [Zukowski et al. 2007], where a query can join the column scan at any partition. The query plan is extended by an optimizer to maximize the opportunities to exploit cooperative actions on the shared data chunks as they become available.

Hence, the ratio between the buffer space and the data size for the workload is less than one, i.e., there is often more buffer space available to start newly arrived queries. The direct data access to the RDMA buffers and the internal query parallelization with cooperative access to the partitions increases the throughput and keeps the latency low.

The intra-query parallelization and the injection of the calls is straight-forward. The DaCy optimizer injects $N$ requests, $N$ pins and $N$ unpins into the query plan for each column. After the $Nth$ *pin()* call the optimizer inserts a *pack()* call. This instruction does the union of all partitions rebuilding the original BAT. For example, Figure 13 depicts the plan of Figure 4b, but each column is now decomposed into three partitions.

One of the distinctive features of MonetDB is to materialize the complete result of each operator. Operation $mat.pack$ gathers the pieces and glues them together to form the final result. Alternatively, the pieces are kept independent to improve parallel dataflow driven computation. Using Figure 4b, Figure 13 exhibits both cases: the instruction *bat.reverse(X9)* is parallelized while the instruction *algebra.join(X1,X9)* is not.

Parallel processing can be further improved by postponing *packing*, i.e., the subsequent instructions are parallelized using the partitioned results. In Figure 13, instruction $B$ could also be parallelized by only doing the *pack* before instruction $C$. There exists a plethora of alternative query plan optimizations, whose description is beyond the scope of this paper.

### 5.3. Buffer management.

The DaCy storage layer consists of several buffers and each has a property list to determine the used/free space and if they contain data *in transit* or data *in use* by the DBMS. They are used by the DaCy buffer management to determine the number of buffers for hot-set propagation and to properly feed the local starving queries. Global and local throughput is dependent on the success of these management decisions.

The DaCy storage is divided into the *DaCy space* and *DBMS space*. Reserving just a few buffers for *transit data* slows down data propagation and decreases the global throughput. On the other hand, few buffers to store data for the DBMS degrades local query performance. Therefore, DaCy reserves a limited amount of buffer space for each kind. The remaining space, *neutral zone*, is used for both *DaCy* and DBMS depending on the workload requirements.

For workloads with a small hot-set, the *neutral zone*, is used to cache BATs for the DBMS even after they have been used. Hence, they can be re-used by future queries to boost performance. Furthermore, if BATs are owned by the node, they are readily available to be forwarded upon request and reduce data access latency.

On the other hand, a growing hot-set may consume a large part of the *neutral zone* used as queue for the BATs in transit. Therefore, the DBMS BATs are removed as quickly as possible and new additions to the *neutral zone* are forbidden. The first victims to get evicted from the neutral zone are the *cached* BATs, followed by those *in use* by the DBMS. [Be more clear because it seems you are killing the queries :)] The remainder is removed by increasing the $LOIT_n$ threshold.

## 6. EVALUATION

A fully functional prototype is constructed to complement and exercise the protocols designed and analyzed through simulation. The primary goal is to identify advantageous scenarios and possible bottlenecks of the Data Cyclotron approach. The prototype was realized by integration of the

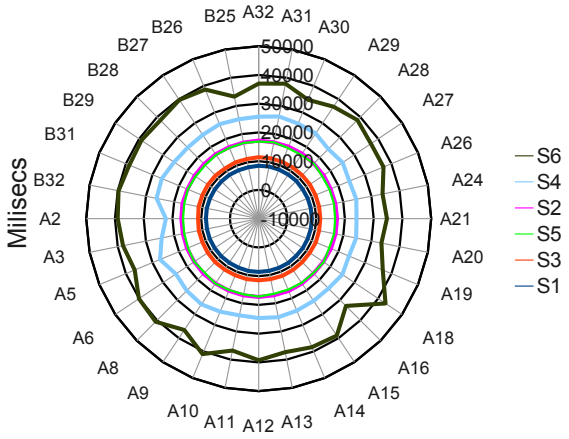



Fig. 14: Query Average Time per node.

software components with MonetDB [9]. The experiments are not intended to benchmark our novel architecture against contemporary approaches of single/parallel solutions. The engineering optimization space for the Data Cyclotron is largely unexplored and experience gained to date with the architecture would not do justice to such a comparison.

The evaluation was done on a computer cluster from our national super-computer center [10] equipped with 256 machines, all of them interconnected using a Qlogic 4x DDR InfiniBand card, i.e., the bandwidth is 1600 MB/sec and the latency below 6 $\mu$sec. Each machine is equipped with 2 Intel quad-core Xeon L5520 (2.26GHz clock), 24GB main memory, 8 MB cache size, 65GB scratch space (200 MB/sec read access) and 5.86 GT/s (GigaTransfers per second) quick path interconnect. Each rack has two switches where each of them has 16 nodes directly interconnected. Evidently, such a national infrastructure poses limitations on the resources available to individual researchers for experimentation. Here we report on the experiences gained to date and provide an outlook on the challenges to be addressed by followup research.[11]

In the first round of micro-benchmarking experiments we show the influence of the network type, the load on the ring and variations in the ring size on the data access behavior. It is followed with an analysis of the system throughput using a full-scale TPC-H implementation with workloads of thousands of queries and hundreds of concurrently running queries. In both cases, the issues stemming from rings spanning multiple racks are exposed and the solutions for low data-access latency and high throughput are presented.

### 6.1. Data Access Latency.

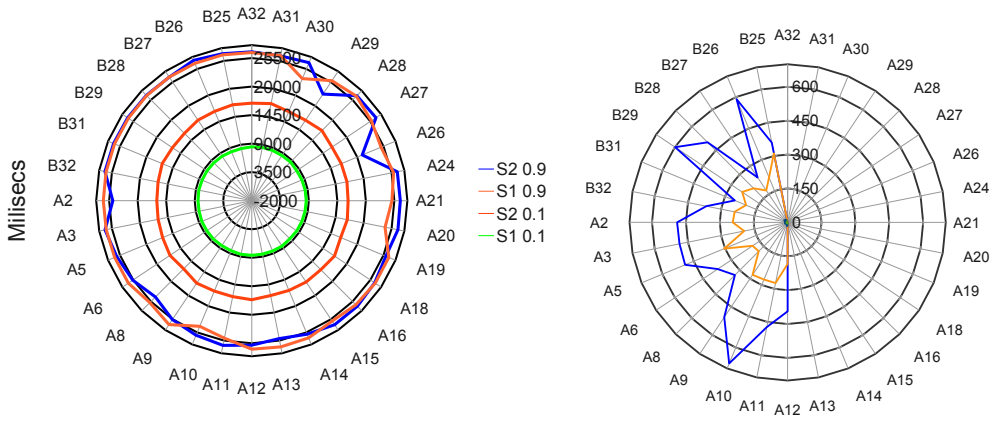
The base topology for the first experiment was a ring composed of thirty-two nodes. The nodes are spread among two racks, rack $A$ and $B$. The nodes are interconnected in them same order as the one represented in Figure 14.

All nodes dedicate one third of their memory for RDMA buffers, *DaCy storage*, leaving enough memory space for intermediate query results. Hence, the query performance is not affected by memory BUS contention, nor being bounded by the local disk bandwidth.

[9]For the integration, we added a new optimizer to the optimizers stack, called *opt_DataCyclotron* available in MonetDB release Oct2010

[10]http://www.sara.nl/

[11]A dedicated 300 database machine for further exploration of the solution space and detailed analysis is commissioned for the fall of 2011. http://www.scilens.org/platform




(a) Query Average Time for low and high LOITn.

(b) Number of Loads per node.

Fig. 15: Hot-Set Management.

The micro-benchmark data-set consists of three 64-column wide tables, $TA$, $TB$, and $TC$. They differ in the number of partitions (chunks) per column: $TA$ uses 2 partitions, $TB$ 3 partitions, and $TC$ 6 partitions. Each partition is 128 MB to optimize network transport, leading to a total database size of around 270 GB.

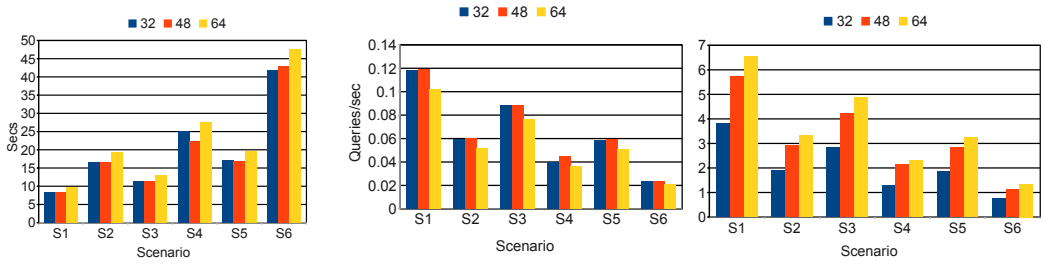
The micro benchmark workload is divided into 6 scenarios, $S1, .., S6$. Each contains 1024 simple select queries over 11 randomly chosen columns, 4 joins and a projection on 3 columns all from the same table. The first two scenarios, $S1$ and $S2$ use $TA$, $TB$ is used by $S3$ and $S4$, and $TC$ is used by $S5$ and $S6$. To have different hot-set sizes, $S1$, $S3$, and $S5$ only touch on half of the columns, while the remaining scenarios, $S2$, $S4$, and $S6$, touch on all the columns.

Furthermore, to stress the Data Cyclotron buffer pool for data access and forwarding, each query keeps 70 to 90% of the requested data in the buffers, i.e., DBMS space load, and the $LOITn$ is kept as low as possible to maximize the amount of hot-set data in the ring, i.e., the DaCy space load. Hence, from scenario 1 up to 6, the DBMS space is increased shrinking the DaCy space, i.e., the available space to store the hot-set.

All workloads are executed with $LOITn$ below 0.3 with the exception of $S6$ where $LOITn$ reached the value 0.9 due to the lack of space in the ring storage to accommodate the hot-set, which aligns with the observations in Section 4.1.1. The queries are flowing in the ring and nodes automatically pick and execute one query at a time. The query execution time average is pictured in Figure 14. For different hot-set sizes, Data Cyclotron was able to provide the required data for the queries at high speed. The average time increase is close to linear for the ones with the low $LOITn$, i.e., the complete hot-set is kept in the ring.

Despite buffer contention for data access and forwarding, the data stream achieved 1.3 GB/s in the best cases and 0.9GB/s in the worst cases. Since the DDR uses 8B/10B encoding, $1.6 \times 0.8 \approx 1.3GB/s$. It demonstrates how well the buffer space is managed between the Data Cyclotron and the DBMS. Unfortunately, the 1.3GB/s bandwidth is only achieved within the same hardware rack and network switch. The ratio between the hot-set size and the query execution time average shows that for each extra GB of hot-set data, the execution time increases between 750 msecs and 950 msecs, i.e., it is bounded by the bandwidth.

The inherent data flow, i.e., the intra-query parallelism, makes it possible to access all data needed within a single Data Cyclotron cycle. Hence, for I/O bounded workloads the query time execution is bounded by the hot-set size, network bandwidth and buffer space.




(a) Query Execution Average per Ring (b) Throughput per Node (c) System Throughput

Fig. 16: Different Rings.

Data Cyclotron is, however, intended for heavy and complex analytic queries composed of instructions with execution time reaching a few hundreds of seconds. This creates an interval where data access overlaps with data processing and, due to RDMA, without taking CPU cycles.

**Hot-set management**

Nonetheless, those workloads scan a big portion of the database. With a large hot-set, the storage ring quickly becomes overloaded and stale data has to be unloaded. The latency is higher for this data access pattern. For it is directly affected by its distribution among the nodes and the load in the ring storage.

The impact of eviction from the Data Cyclotron buffer is analyzed using an experiment with two different hot-sets using a low (0.1) and high (0.9) value for $LOITn$. The experiment uses unbalanced distribution of the data, i.e., the data is stored in half of the ring.

For a low $LOITn$ only the hot-set size influences the query execution time, since all data needed is flowing in the ring. For a high $LOITn$ the execution time for both hot-sets converge to the average time needed to request and receive data. The average time for a cycle in our experimental setup is 8 seconds, but due to the buffer contention at the busiest nodes, data load is postponed often. The number of chunks loaded per node is depicted in Figure 16. The nodes of the left side of the ring handle the data for local query processing, the flow of requested data from its neighbors and still load data from their local disks into the storage ring. The buffer overload forces them to increase the local $LOITn$ to higher levels to remove more data from circulation.

The workload used in this experiment capitalizes on the degradation of the data access latency. Contrary to a non-uniform workloads, such as the *Gaussian Workload* in Section 4.1.3, for which the *in vogue* chunks are kept in the storage ring while the *unpopular* chunks are unloaded to release buffers for pending loads, in a uniform workloads the chunks have more or less the same $LOI$. Therefore, all of them have the same probability of being unloaded. To cope with this issue, the hot-set management should have a priority policy based on the requests flow. Hence, the nodes should keep track of the number requests absorbed while the data was in circulation, or pending for loading, to then determine a load priority and at the *UpdateLoi* time, Figure 8, estimate the new LOI for a more precise decision to cold down, or not, a chunk.

**Ring Extension**

Any system configuration faces the point that it lacks resources to accommodate a timely execution of all queries. In the Data Cyclotron this can be partly alleviated using a ring extension. With the same setup, the ring is extended twice through insertion of 16 nodes: $R1$, $R2$, and $R3$. The insertion happens on the right side of the ring to keep the un-balanced distribution of data.

The results of this experiment are shown in Figure 16. For each scenario the average time for each query remains almost the same as for the first extension. In the second extension the average time over all scenarios increased between 10 and 16%. With up to 48 nodes, the throughput per node increased slightly. However, for rings beyond 64 nodes we reach a saturation point and the

throughput per node decreased by 13%. The length of the ring also leads to a higher data latency, but since the hot-set size did not change and with a high bandwidth being available, the system throughput increased by 70% when 16 nodes were added. Addition of another 16 nodes only brought 14% improvement (except for $W5$). It indicates that the *saturation point* can be reached by another extension of 16 nodes. The dynamic adaptation of the ring topology to cope with the phenomenon is part of ongoing work *Pulsating Rings* and explained in Section 7.3.

### 6.2. Throughput.

The Data Cyclotron is designed for applications with complex analytic queries and high throughput demands. The following study demonstrates how to exploit its features using TPC-H as a frame of reference. It also identifies some boundaries of the architecture imposed by the hardware configuration and how they could be resolved when a system is assembled from scratch.

The first claim of the Data Cyclotron is that remote memory is faster than access to a local disk. In the same line as the micro-bench mark experiments, i.e., ring sizes and a hot-set size, we show how access to remote memory outperforms a single node disk access.

Using TPC-H scale factor 40 and executing all the queries in sequence at one of the nodes, the boosted time for execution is evident. The query execution time improvement stems from the distributed data load using all nodes. Complex queries that require a scan of the complete lineitem table, like Q1, Q5 and Q21, profit most from the parallel data load as represented on Table II.

| | 1 | 32 | 48 | 64 |
|---|---|---|---|---|
| 1 | 208.4 | 11.5 | 14.1 | 14.6 |
| 2 | 13.6 | 3.6 | 4.9 | 4.8 |
| 3 | 14.7 | 8.3 | 8.6 | 10.9 |
| 4 | 30.4 | 4.8 | 5.4 | 5.8 |
| 5 | 16.9 | 7 | 8.6 | 10.1 |
| 6 | 3.4 | 5.8 | 5.3 | 6.3 |
| 7 | 7.6 | 10.6 | 13.7 | 17.6 |
| 8 | 20.6 | 10 | 9.9 | 11.1 |
| 11 | 11.3 | 3.9 | 5.9 | 4.6 |
| 12 | 15.2 | 5.6 | 7.4 | 7.7 |
| 14 | 4.9 | 5.1 | 5.6 | 6.3 |
| 15 | 13.9 | 12 | 12.8 | 12.6 |
| 17 | 36.4 | 37.9 | 73.4 | 74.9 |
| 18 | 125.2 | 78.9 | 108.1 | 112.8 |
| 19 | 79.3 | 22.6 | 28.3 | 28.8 |
| 20 | 11 | 7.6 | 7.7 | 8.2 |
| 21 | 31.8 | 10.2 | 15.8 | 16.5 |
| 22 | 14.9 | 8.11 | 9.6 | 10.2 |

Table II: Query Execution Times

For ring $R1$, there are 32 nodes reading data concurrently. It behaves like a networked storage system with 32 disks with access bandwidth near to 1.3 GB/s, not far from the 2GB/s bandwidth achievable by an expensive state of the art RAID system with 4 SSD disks. However, if such a system is not integrated with DMA channels, CPU cycles will be shared between data transfer and data processing. With Data Cyclotron, and as a result of the RDMA benefits, Section 2.3.2, the CPU was fully devoted to data processing. Therefore, remote data access is overlapping with data processing, which explains the constant gain for $R2$ and $R3$.

With the data flowing though the distributed ring buffer composed of remote memories, the queries can be distributed among the nodes without affecting too much the latency. It definitely increases system throughput. To illustrate we used a workload composed by 1024 queries derived from the TPC-H queries using an uniform distribution. As in 6.1 each node peaks and executes one query at the time.

The results in Table III support our intuition such as throughput improvement. Although, the query execution per query was not as optimal as expected Figure 17. A close look at the query trace pointed towards a less than optimal exploit of the continuous data stream. The current execution model is still pure pull-based where the plan is static and the data is retrieved as the instructions are executed. The intra-query parallelization gave some degree of flexibility, but not enough to optimally exploit the continuous stream of data. It seems that the query execution model underlying MonetDB should adapt more to the arrival order of the partitions passing by. A dynamic adaptation will reduce the data access latency and the number of buffers loaded with the data waiting for processing. The realization and quantification of this approach remains a topic for future work.

| Nodes | 1 | 32 | 48 | 64 |
|---|---|---|---|---|
| Throughput | 3 | 5 | 7 | 9 |

Table III: Throughput

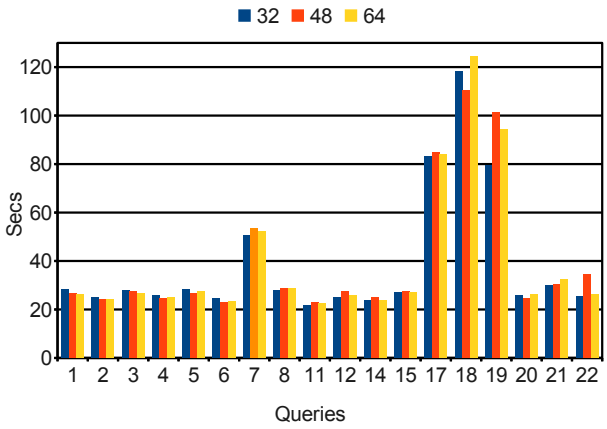



Fig. 17: Query Average Time With All Nodes.

### 6.3. Big Rings.

Big rings brought new challenges to the Data Cyclotron implementation.

The latency to access the less popular chunks and fresh chunks may become a performance hindrance. As already observed in the Broadcast disks approach [Acharya et al. 1995], the optimal solution to reduce latency in a ring topology ranges from boosting the speed for the most popular data, i.e., increasing their frequency in the broadcast stream, and to cache the highly used ones with lowest frequency. We introduce some changes to the DataCyclotron architecture to achieve the same goal, although, we take another course.

*6.3.1. Data Forwarding.* The Data Cyclotron ring increases the speed of the most popular chunks by priority based forwarding instead of FIFO order. The priority is determined by its actual LOI and the number of possible touches in the next nodes. The latter is determined during the request trip through the ring. Each request collects and shares statistics while traveling. Those statistics are then used to define the load priority

at the chunk's owner and during its journey in the ring. In particular, the forwarding thread selects the chunks for forwarding using a Zipfian distribution instead of a FIFO selection. The process is analog, to an highway, where several lines co-exist with different speed limits, i.e., levels of priority.

The analogy brings to light the use of shortcuts to reduce the time for a popular chunk to reach a distant node in the ring. It reduces the latency for the most popular chunks. Although, since the data will only be available at a subset of the nodes, it could lead to throughput degradation, especially for skewed workloads.

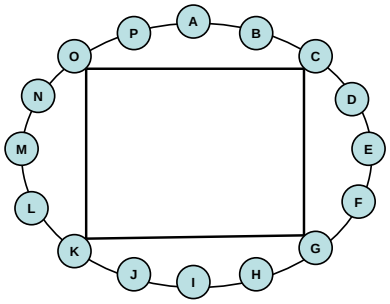



Fig. 18: Internal Ring.

To circumvent the problem, we introduce an inner ring represented as a square in Figure 18. The vital chunks now flow within the internal ring. Each of the nodes from this ring can fulfill the requests from the nodes ahead of them. For example, node $O$ will fulfill the requests from all nodes between $O$ and $C$. In case $C$ does not have the chunk it will send a request within the internal ring.

*6.3.2. Data Caching.* To benefit from the new forwarding scheme, the Data Cyclotron has to adapt its cache policy to reduce data latency. The chunks, are not flowing anymore with the same *speed*, i.e., their cycle time depends on their priority. Therefore, caching the used chunks, because they are most likely to be accessed in the future, should be replaced by a caching policy similar to the one used in [Acharya et al. 1995].

In [Acharya et al. 1995] , pure pushed-based system, the authors demonstrated that under certain assumptions, the clients should not cache their hottest pages, but rather, they should store pages for which the local probability of access is significantly greater than the pages frequency in the broadcast stream, i.e., pages that are popular at one client, but unpopular in the remain clients.

Hence, based on the local utilization and the forwarding priority, the Data Cyclotron determines which chunks are stored or evicted. By caching the less popular chunks, the Data Cyclotron reduces the data access latency for them and the same time, the most popular chunks are not slowed down by their forwarding. The trade offs of this policy will be study in detail in future publication.

## 7. FUTURE OUTLOOK

The Data Cyclotron outlined is an innovative architecture with a large scope for scientific challenges. In this section, the most prominent areas are scratched upon, e.g., query processing, result caching, structural adaptation to changes in the workload, and updates.

### 7.1. Query Processing

A query is not bound to stay and be executed at the node where it enters the Data Cyclotron. In fact, the ultimate goal is to achieve a self-organizing, adaptive load balance using the aggregated autonomous behavior of each node. Hence, once the data chunk requests are sent off, a query can start with a nomadic phase, "chasing" the data requests upstream to find a more satisfactory node to settle for its execution. At each node visited, we ask for a bid to execute the query locally. The *price* is the result of a heuristic cost model for solving the query, based on its data needs and the node's current workload.

In addition, the Data Cyclotron architecture allows for highly efficient shared-nothing *intra*-query parallelism. During the nomadic phase, a query can be split into independent sub-queries to consume disjoint data subsets at different nodes. The number of sub-queries depends on the price attached dynamically. All sub-queries are then processed concurrently, each settling on a different node following the basic procedures of a normal query. The individual intermediate results are combined to form the final query result.

### 7.2. Result Caching

Multi-query processing can be boosted by reusing (intermediate) query results to avoid (part of) the processing cost, i.e., they are simply treated as persistent data and pushed into the storage ring for interested queries. Like base data, intermediate results are characterized by their age and their popularity on the ring. They only keep flowing as long as there is interest.

The scheme becomes even more interesting when combined with the intra-query parallelism. Then multiple sub-queries originating from a single query in execution create a large flow of intermediate results to boost others. The idea was already tested and implemented for a centralized system in [Ivanova et al. 2009; Ivanova et al. 2010].

There is a plethora of optimization scenarios to consider. A node can throw all intermediate results it creates into the ring. Alternatively, intermediates can stay alive in the local cache of their creator node as long as possible. If a request is issued, they enter the ring, otherwise they are gradually removed from the cache to make room for new intermediates.

### 7.3. Pulsating Rings

The workload is not stable over time. The immediate consequence is that a Data Cyclotron ring structure may not always be optimal in terms of resource utilization and performance. For example, having more nodes than strictly necessary increases the latency in data access. However, having too

few nodes reduces the resources for efficient processing. The challenge is to detect such deviations quickly and to adapt the structure of the ring.

For this, we introduce the notion of *pulsating rings* that adaptively *shrink* or *grow* to match the requirements of the workload, i.e., nodes are removed from the ring to reduce the latency or are thrown back in when they are needed for their storage and processing resources. The decision to leave a ring can be made locally, in a self-organizing way, based on the amount of data and number of requests flowing by the nodes. For example, a node individually decides to leave a ring if its resources are not being exploited over a satisfying threshold.

The updates to the ring are localized to the two (envisioned) node's neighbors. A new node simply jumps into the data flow, between two nodes, Figure 19. A leaving node simply notifies its predecessor node and its successor node to make a direct link and to ignore the leaving node from then on.

The ring never shrinks more than beyond its initial size and it can be extended as long as there are nodes available in the cluster. The new node's functionality is only data forwarding and query processing.

The pulsating rings flexibility can be improved through data delegation, i.e., a node can delegate to, or request from, another node the data ownership. Hence, on a ring extension a node with high rate of BAT loads delegates a percentage of its most popular data to the new node. On a ring contraction a leaving node, before disconnecting, delegates the data under its ownership to its neighbors. Since any node can delegate its data, the ring can shrink beyond its initial structure, i.e., it can shrink until the least number of nodes to assure enough persistent storage for the data set.

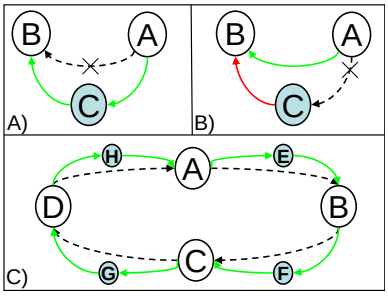



Fig. 19: The insertion and removal of a node

### 7.4. The Space for Updates.

Let us briefly discuss handling updates in this setting. As a first step, we can exploit the fact that a *single* copy of each relation is flowing through a Data Cyclotron as a set of disjoint data chunks. This way, each data chunk can be updated separately by any node without needing to synchronize the process with the other data chunks.

Since a data chunk $C$ can be accessed at any node, only one node can have the lock to update $C$. In the initial phase, i.e., none has the locks, an update query is treated as any other query. It enters the system and searches for a controlling node $N$ to settle and waits for the data chunks to pass by.

The only difference is that when node $N$ processes an update request it propagates the data chunks with a globally unique lock tag. This tag informs that these data chunks ($Cs$) cannot be updated at any other node. Hence, any update query for $Cs$ or sub-group of $Cs$ must enter the ring and hop from node to node until it reaches $N$.

In case a query contains two groups of data chunks locked at different nodes, the query will settle at the node with the higher lock tag. Once settled the node will request the other locks.

A node that receives a lock request from a node with highest tag, must finish the local update queries, send the pending ones into the ring, and release the locks. In case more than one node is competing for the locks, the node with the lowest tag has the highest priory. The effect of updates is that the system becomes polluted with several versions of a data chunks. The maintenance of these versions is similar to as the rest of the database, i.e., using its $LOI$.

The locks are always released, i.e., the tags removed, once the queries have been finished. The read-only queries interested in data chunks with flags refrain from processing them, recognizing their stale state; they have to wait for the new version. Read-only queries that do not necessarily require the latest updated version can continue using the old versions passing by.

The lock locations becomes stable if the update pattern of the queries remains constant over time. However, the location distribution also depends on the input data overlap between queries. With too many overlapping inputs, the distribution is skewed whereas with disjoint inputs the locks are spread among the nodes balancing the resources utilization for update queries.

## 8. RELATED WORK

The Data Cyclotron does not stand alone in the database research. The continuous flow of the hot-set, ring topology, and the data-access protocols to exploit the new hardware trends have been studied for the last decades in several disciplines. Hence, we revisit the milestones influencing our architectural design and highlight related concepts in database research now ready for in-depth re-evaluation. Furthermore, we explain how Data Cyclotron can be smoothly integrated into a row store database engine.

### 8.1. History

Distributed query processing to exploit remote memories and high speed networks to improve performance in OLTP has been studied in [Ioannidis et al. 1998; Flouris and Markatos 1998]. It is also addressed in the area of distributed systems as data broadcasting and multi-cast messaging, such as [Herman et al. 1987; Banerjee and Li 1994; Acharya et al. 1995; Acharya et al. 1997; Aksoy et al. 2001]. Solutions include scheduling techniques at the server side [Acharya et al. 1995], caching techniques at the client side [Acharya et al. 1995], integration of push-based and pull-based broadcasts [Acharya et al. 1997], and pre-fetching techniques [Aksoy et al. 2001]. Most systems ignore the data content and their characteristics [Kitajima et al. 2007]. The seminal DataCycle [Herman et al. 1987] and Broadcast Disks approach [Acharya et al. 1995] are exceptional systems to be looked at more carefully.

**DataCycle.** The DataCycle [Herman et al. 1987; Banerjee and Li 1994] makes data items available by repetitive broadcast of the entire database stored in a central pump. The broadcasted database is filtered on-the-fly by microprocessors optimized for synchronous high-speed search, i.e., evaluation of all terms of a search predicate concurrently in the critical data movement path. It eliminates index creation and maintenance costs. The cycle time, i.e., the time to broadcast the entire database, is the major performance factor. It only depends on the speed of hardware components, the filter selectivity, and the network bandwidth.

The Data Cyclotron and DataCycle have some commonalities, however they differ on four salient points. The Data Cyclotron uses a pull model to propagate the hot set for the query workload only. It does not distinguish the participants, all nodes access the data and contribute data, i.e., there is no central pump. Each node bulk loads database content structured by a relational scheme and not as a tuple stream. Finally, the Data Cyclotron performance relies on its self-organizing protocols which exploit the available resources, such as ring capacity and speed. These self-organizing protocols are key for the success of our architecture compared to DataCycle.

A distributed version of DataCycle [Banerjee and Li 1994] has been proposed in [Bowen et al. 1992] and studied in [Banerjee and Li 1994]. They propose a distributed scheme as a solution for the lack of scalability of the central scheme. Moreover, they focus on the data consistency when several pumps are used. The optimization of the broadcast set, to reduce the latency, is pointed out, but not studied in detail.

**Broadcast Disks.** The Broadcast Disk [Acharya et al. 1995] superimposes multiple disks spinning at different speeds on a single broadcast channel creating an arbitrarily fine-grained memory hierarchy. It uses a novel multi-disk structuring mechanism that allows data items to be broadcasted non-uniformly, i.e., bandwidth can be allocated to data items in proportion to their importance. Furthermore, it provides client-side storage management algorithms for data caching and pre-fetching tailored to the multi-disk broadcast.

The Broadcast Disk is designed for asymmetric communication environments. The central pump broadcasts according to a periodic schedule, in anticipation of client requests. In later work a pull-

back channel was integrated to allow clients to send explicit requests for data to the server. However, it does not combine client requests to reduce the stress on the channel.

In [Acharya et al. 1997] the authors address the threshold between the pull and the push approach. For a lightly loaded server the pull-based policy is the preferred one. However, the pure push-based policy works best on a saturated server. Using both techniques in a system with widely varying loads can lead to significant degradation of the performance, since they fail to scale once the server load moves away from their optimality niche. The IPP algorithm, a merge between both extremes pull- and push-based algorithm, provided reasonably consistent performance over the entire spectrum of the system load.

The Data Cyclotron differs in two main aspects, a) it is not designed for asymmetric communication environments and b) it does not have a central pump. All nodes participate in the data and requests flow. Furthermore, requests are combined to reduce the upstream traffic. Therefore, we do not encounter problems using a pure push model which, according to [Acharya et al. 1997], is suited for systems with dynamic loads. Finally, for data propagation, the bandwidth is used uniformly by the data items, i.e, we do not have a multi-disk structuring mechanism.

### 8.2. Related Concepts

The Data Cyclotron follows a push model, the data is sent to the node's memory for processing. This abstraction is also used by *"data-centric"* applications, such as DataPath [Arumugam et al. 2010]. A similar abstraction exists between *Cooperative Scans* [Zukowski et al. 2007] and the data access by the queries, Section 5.2.

A further shared concept is the use of a group of inter-connected nodes to perform distributed query processing. The P2P and the Grid architectures have attempted to create efficient and flexible solutions.

**DataPath.** Similar to Data Cyclotron, DataPath explores the continuous data flow between the memory and the CPU Cores to amortize the data access costs among every computation that uses the data.

In DataPath, queries do not request data. Instead, data is automatically pushed onto processors, where they are then processed by any interested computation. If there is no spare computation to process the data, the data is simply dropped.

All query processing in the DataPath system is centered around the idea of a single, central path network. A DataPaths path network is a graph consisting of data streams (called paths) that force data into computations (called waypoints).

Contrary to Data Cyclotron, the DataPath is designed for a single machine. It is bounded by the local number of cores and the main-memory size, i.e., it cannot scale. Furthermore, it needs dozens of disks to optimize the data load into main-memory, i.e, the data partition and distribution is driven by the workload type.

Being a pure push-based model, workload shifts or queries with dissimilar plans complicate the adaptation of the path network, i.e., the system is not as flexible as Data Cyclotron.

**Cooperative Scans.** We would like to relate our work to *Cooperative Scans*[Zukowski et al. 2007]. Cooperative Scans exploits the observation that concurrent scans have a high expected amount of overlapping data need. They explore the opportunity to synchronize queries, such that they share buffered data and thus reduce the demand for disk bandwidth.

The synchronization uses a relevance policy to prioritize starved queries. It tries to maximize buffer pool reuse without slowing down fast queries, i.e., it optimizes throughput and minimizes latency.

These queries are prioritized according to the amount of data they still need, with shorter queries receiving higher priorities. Furthermore, to maximize sharing, chunks that are needed by the highest number of starved queries are promoted, while at the same time, the priorities of chunks needed by many non-starved queries are adjusted slightly.

The Data Cyclotron and Cooperative scans share the goal to improve throughput and minimize latency. However, the Data Cyclotron was designed for distributed query processing and explores

the access to remote memory instead of to disk. Although, some of the techniques and observations made in [Zukowski et al. 2007] may still be applicable to *Data Cyclotron*.

**P2P and Grids.** A more recent attempt to harness the potentials of distributed processing are the P2P and the Grid architectures. They are built on the assumption that *a node is statically bound to a specific part of the data space*. This is necessary to create a manageable structure for query optimizers to derive plans for queries and data to meet. However, a static assignment also complicates reaching satisfactory load balancing, in particular with dynamic workloads, i.e., at any time *only part of the network is being used*. For example, in a Distributed Hash Table (DHT), such as [Ratnasamy et al. 2001; Stoica et al. 2001], each node is responsible for all queries and tuples whose hash-values fall within a given range. The position of a node in the overlay ring defines this range. This is a significantly more dynamic approach as a node can, in a quick and inexpensive way, change its position in the DHT. It effectively transfers part of its load to another node. The main difference with the Data Cyclotron is our reliance on a full fledged database engine to solve the queries, and shipping large data fragments around rather then individual tuples selected by their hash value.

### 8.3. 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 MonetDB prototype showed how easy it was to integrate the Data Cyclotron software. In this section we consummate such fact with a description of how Data Cyclotron could be integrated with a typical row-store DBMS.

A generic row store DBMS is composed of many components, but it is at the backend where iteration over the data takes place. In these systems, the usual query life-cycle 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. It 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 execution of the query 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 20. 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 result of the query.

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 up front about all the 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.

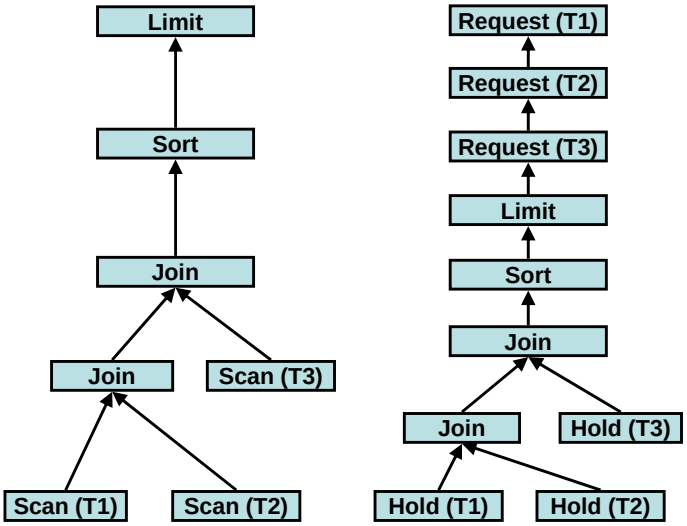



Fig. 20: Typical execution plan in row-store DBMS

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-partition.

**Data Partition.**

The 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 Data Cyclotron 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 MAP-REDUCE, 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. However, the discussion of this optimization is out of scope for this paper.

**Extended work.**

This manuscript extends the work presented in [Goncalves and Kersten 2010] by analyzing the collaborative interaction between the Data Cyclotron with the network and the DBMS. For the network interaction, it decomposes the components of the DaCy architecture and exposes its advantages for the integration with the state-of-the-art in networking to achieve low latency and high throughput.

For the DBMS interaction, it is shown how the data access provided by DaCy can be exploited by a column store with intra-query parallelization. Furthermore, the integration of a generic row-store architecture is discussed and briefly described.

Finally, the work is concluded with the conception of the Data Cyclotron architecture, i.e., implementation of a prototype in an Infiniband cluster. Its evaluation showed its alignment with the simulation results in [Goncalves and Kersten 2010] and its ability to scale up and speedup.

## 9. CONCLUSIONS

The Data Cyclotron architecture is a response to the call-to-arms in [Kersten 2008], which challenges the research community to explore novel architectures for distributed database processing.

The key idea is to turn data movement between network nodes from being an evil to avoid at all cost into an ally for improved system performance, flexibility, and query throughput. To achieve this goal the database hot set is continuously moved around in a closed path of processing 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 to changes in the workload set quickly.

The protocols were designed and improved through a network simulator. A full function prototype was constructed to complement them and analyze the performance consequences using a micro-benchmark and TPC-H on Infiniband cluster.

They confirm our intuition that a storage ring based on the hot set can achieve high throughput and low latency. Moreover, the experiments show the robustness of the setup under skewed workloads.

The paper opens a vista on a research landscape of novel ways to implement distributed query processing. Cross fertilization from distributed systems, hardware trends, and analytical 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.

## ACKNOWLEDGMENTS

We wish to thank the members of the MonetDB team for their continual support in the realization of the Data Cyclotron. We thank the reviewers for their constructive comments and guidance to improve the depth and quality of our earlier version presented at EDBT2010. We also thank Willem Vermin's team for the support provided on the LISA cluster at the High Performance Computing Center of Netherlands.

## REFERENCES

ACHARYA, S., ALONSO, R., FRANKLIN, M., AND ZDONIK, S. 1995. Broadcast Disks: Data Management for Asymmetric Communication Environments. In *SIGMOD '95*. 199–210.

ACHARYA, S., FRANKLIN, M., AND ZDONIK, S. 1997. Balancing push and pull for data broadcast. In *SIGMOD '97*. 183–194.

AKSOY, D., FRANKLIN, M. J., AND ZDONIK, S. B. 2001. Data staging for on-demand broadcast. In *VLDB '01*. 571–580.

ARUMUGAM, S., DOBRA, A., JERMAINE, C. M., PANSARE, N., AND PEREZ, L. 2010. The datapath system: a data-centric analytic processing engine for large data warehouses. In *SIGMOD '10*. ACM, New York, NY, USA, 519–530.

BALAJI, P. 2004. Sockets vs rdma interface over 10-gigabit networks: An in-depth analysis of the memory traffic bottleneck. In *In RAIT workshop 04*.

BANERJEE, S. AND LI, V. O. K. 1994. Evaluating the distributed datacycle scheme for a high performance distributed system. *Journal of Computing and Information 1*.

BONCZ, P., KERSTEN, M., AND MANEGOLD, S. 2008. Breaking the Memory Wall in MonetDB. *Commun. ACM 51,* 12, 77–85.

BONCZ, P., MANEGOLD, S., AND KERSTEN, M. 2002. Optimizing Main-Memory Join On Modern Hardware. *IEEE Transactions on Knowledge and Data Engineering 14,* 4, 709 – 730.

BOSE, S. K., KRISHNAMOORTHY, S., AND RANADE, N. 2007. Allocating resources to parallel query plans in data grids. In *GCC '07*. 210–220.

BOWEN, T. F., GOPAL, G., HERMAN, G., HICKEY, T., LEE, K. C., MANSFIELD, W. H., RAITZ, J., AND WEINRIB, A. 1992. The datacycle architecture. *Commun. ACM 35,* 12, 71–81.

CHAUDHURI, S. AND NARASAYYA, V. R. 2007. Self-tuning database systems: A decade of progress. In *VLDB '07*. 3–14.

CHUN, T. 2001. Performance studies of high-speed communication on commodity cluster. Thesis.

DEWITT, D. AND GRAY, J. 1992. Parallel Database Systems: The Future of High Performance Database Systems. *Communications of the ACM 35,* 6, 85–98.

FLOURIS, M. D. AND MARKATOS, E. P. 1998. The network ramdisk: Using remote memory on heterogenous nows.

FOONG, A., HUFF, T., HUM, H., PATWARDHAN, J., AND REGNIER, G. 2003. TCP performance re-visited. In *Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software*. 70–79.

FREY, P., GONCALVES, R., KERSTEN, M., AND TEUBNER, J. 2009. Spinning relations: High-speed networks for distributed join processing. In *DaMoN '09*. 27–33.

FREY, P., GONCALVES, R., KERSTEN, M., AND TEUBNER, J. 2010. A spinning join that does not get dizzy.

FREY, P. W. AND ALONSO, G. 2009. Minimizing the hidden cost of RDMA. In *ICDCS '09*. 553–560.

GONCALVES, R. AND KERSTEN, M. 2010. The data cyclotron query processing scheme. In *EDBT '10*.

GOVINDARAJU, N., GRAY, J., KUMAR, R., AND MANOCHA, D. 2006. Gputerasort: High performance graphics co-processor sorting for large database management. In *SIGMOD '06*. 325–336.

GRAY, J., SZALAY, A. S., ET AL. 2002. Data Mining the SDSS SkyServer Database. *MSR-TR-2002-01*.

HABABEH, I. O., RAMACHANDRAN, M., AND BOWRING, N. 2007. A high-performance computing method for data allocation in distributed database systems. *The Journal of Supercomputing*, 3–18.

HERMAN, G., LEE, K. C., AND WEINRIB, A. 1987. The datacycle architecture for very high throughput database systems. *SIGMOD Rec. 16,* 3, 97–103.

IOANNIDIS, S., MARKATOS, E. P., AND SEVASLIDOU, J. 1998. On using network memory to improve the performance of transaction-based systems.

IVANOVA, M., KERSTEN, M., NES, N., AND GONCALVES, R. 2010. An Architecture For Recycling Intermediates In A Column-Store. *ACM Transactions on Database Systems 35,* 4.

IVANOVA, M., KERSTEN, M., NES, N., AND GONÇALVES, R. 2009. An architecture for recycling intermediates in a column-store. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*. ACM, New York, NY, USA, 309–320.

IVANOVA, M., NES, N., GONÇALVES, R., AND KERSTEN, M. L. 2007. MonetDB/SQL Meets SkyServer: the Challenges of a Scientific Database. In *Proc. SSDBM*. Banff, Canada.

KERSTEN, M. L. 2008. The Database Architecture Jigsaw Puzzle. In *ICDE '08*. 3–4.

KITAJIMA, S., TERADA, T., HARA, T., AND NISHIO, S. 2007. Query processing methods considering the deadline of queries for database broadcasting systems. *Syst. Comput. Japan 38,* 2, 21–31.

KOSSMANN, D. 2000. The state of the art in distributed query processing. *ACM Comput. Surv. 32,* 4, 422–469.

LEE, M. L., KITSUREGAWA, M., OOI, B. C., TAN, K.-L., AND MONDAL, A. 2000. Towards self-tuning data placement in parallel database systems. *SIGMOD Rec. 29,* 2, 225–236.

MANEGOLD, S., KERSTEN, M., AND BONCZ, P. 2009. Database Architecture Evolution: Mammals Flourished Long Before Dinosaurs Became Extinct. In *Proceedings of the International Conference on Very Large Data Bases (VLDB, 2009)* first Ed. VLDB. 10-year Best Paper Award for Database Architecture Optimized for the New Bottleneck: Memory Access. In Proceedings of the International Conference on Very Large Data Bases (VLDB), pp 54-65, Edinburgh, United Kingdom, September 1999.

MÄRTENS, H., RAHM, E., AND STÖHR, T. 2003. Dynamic query scheduling in parallel data warehouses. *Concurrency and Computation: Practice and Experience 15,* 11-12, 1169–1190.

OUSTERHOUT, J., AGRAWAL, P., ERICKSON, D., KOZYRAKIS, C., LEVERICH, J., MAZIRES, D., MITRA, S., NARAYANAN, A., ET AL. 2010. *The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM*. Stanford University.

RATNASAMY, S. ET AL. 2001. A Scalable Content-addressable Network. In *SIGCOMM '01*. 161–172.

STOICA, I. ET AL. 2001. Chord: A Scalable P2P Lookup Service for Internet Applications. In *SIGCOMM '01*. 149–160.

SZALAY, A. S., GRAY, J., ET AL. 2002. The SDSS SkyServer: Public Access to the Sloan Digital Sky Server Data. In *SIGMOD*. 570–581.

YU, C. T. AND CHANG, C. C. 1984. Distributed Query Processing. *ACM Computing 16,* 4.

ZILIO, D. C., RAO, J., LIGHTSTONE, S., LOHMAN, G., STORM, A., GARCIA-ARELLANO, C., AND FADDEN, S. 2004. Db2 design advisor: Integrated automatic physical database design. In *VLDB '04*. 1087–1097.

ZUKOWSKI, M., HÉMAN, S., NES, N., AND BONCZ, P. 2007. Cooperative scans: dynamic bandwidth sharing in a dbms. In *VLDB '07*. 723–734.

Received R; revised e; accepted c
eived November 2010; revised June 2011; accepted August 2011