The Data Cyclotron Query Processing Scheme

R. Goncalves, M. Kersten
CWI, Amsterdam, The Netherlands
{R.A.Goncalves,Martin.Kersten}@cwi.nl

ABSTRACT

Distributed database systems exploit static workload characteristics to steer data fragmentation and data allocation schemes. However, the grand challenge of distributed query processing is to come up with a self-organizing architecture, which exploits all resources to manage the hot data set, minimize query response time, and maximize throughput without global co-ordination.

In this paper, we introduce the Data Cyclotron architecture which addresses the challenges using turbulent data movement through a storage ring built from distributed main memory capitalizing modern remote-DMA facilities. Queries assigned to individual nodes interact with the Data Cyclotron by picking up data fragments continuously flowing around, i.e., the hot set.

Each data fragment carries a *level of interest* (LOI) metric, which represents the cumulative query interest as the fragment passes around the ring multiple times. A fragment with a 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 distributed manor based on ring characteristics and query needs. It optimizes the resource utilization keeping the average data access delay low.

The proposed architecture has a modest impact on existing query execution engines. This is illustrated using an extensive validated simulation study for the Data Cyclotron protocols. The results underpin their robustness in turbulent workload scenarios as well as in the TPC-H scenario. Furthermore, we think that using state-of-the-art network technology, e.g., RDMA, could lead to even more promising results.

The Data Cyclotron architecture opens a new vista for modern distributed database architectures with a plethora of research challenges barely scratched upon.

1. INTRODUCTION

The motivation for distributed query processing has always been to exploit a large resource pool; n nodes can potentially handle a larger workload more efficiently than a single node. It has been a focal area of database research for more than three decades. 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. [11, 22, 27]. Their architectural design is focused on two orthogonal, yet intertwined issues: data allocation and workload behavior.

Sticky Data. Most architectures are based on the premises that a node is made a priori responsible for a specific part of the database using 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 exploit resources optimally. Sticky data calls for a predictable workload to optimally use the system.

Workload Behavior. Workload behavior is, however, often not predictable. Therefore, an active query monitoring system is needed, a database design wizard [9, 28, 23] to advice on indices and materialized views, and followed up with scheduled database maintenance actions. The problem is magnified in the presence of skewness in the query workload and 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. This leads to a short retention period for data- and workload- allocation decisions. Resource utilization may quickly deteriorate.

Thinking Outside the Box. The ultimate goal in this field is to design a self-organizing architecture, which maximizes resource utilization without global coordination, even in the presence of skewed and volatile workloads [9].

There is certainly place to design new data allocation functions [17], grid-base algorithms [8], distributed optimization techniques and associated workload scheduling policies [24]. 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 up to a certain point regarding improved resource utilization. We may end up in a local optimum without a clear view how to find a better one.

On the other hand, hardware trends are on our side, which makes massive processing, huge main memories, and fast interconnects affordable for experimentation with novel architectures [16].

Therefore, in the spirit of an earlier attempt on this field [18, 6], we propose the *Data Cyclotron* architecture, a different approach to distributed query processing by reconsidering de-facto guidelines

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EDBT 2010, March 22–26, 2010, Lausanne, Switzerland.
Copyright 2010 ACM 978-1-60558-945-9/10/0003 ...$10.00

of the past [20]. It provides an outlook on a research landscape barely explored.

Remote Memory at Your Finger Tips. A key observation is that most distributed database systems from the past concentrate on reducing network cost. This was definitely a valid approach for the slow network connections of the past. However, with the continuous advancement of technology it is time to reconsider this approach. 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. Receiving 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 400 MB/s.

While it is widely available in compute clusters, it continues a bit unknown for the database community. Its functionality and appliance as well as the benefits of using such modern hardware are summarized in section 2.

Turbulent Data. With the outlook of RDMA and fast interconnects, we designed the Data Cyclotron 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, which is used to send the *database hot set* continuously around[2]. A query can be executed at any node in the ring; all that it has to do is to announce its interest and to wait for the data 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 the recent hardware trends call for such fresh look and an assessment of the opportunities they create.

Load Balancing. In the Data Cyclotron a node is not assigned to any specific responsibility other than to manage hot data in memory and cold data on its attached disks. Instead, each query searches a lightly loaded node to execute; 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. This innovative and simple strategy avoids *hot spots* that result from errors in the data allocation and query plan algorithms.

Continuous Self-organization. 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.

Simplifying Optimizers. The Data Cyclotron design characteristics affect the query optimizers significantly. Less information is available to select an optimal plan, i.e., the whereabouts of data is not a priori known, nor 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, their frequency of storage ring occupancy, and possible replicas flowing around.

The remainder of this paper is organized as follows. Section 2 introduces RDMA in more detail. Section 3 provides a short introduction of the database engine used in our prototype. Section 4 describes in detail the architecture and algorithms of the Data Cyclotron, Section 5 provides an in depth analysis of the proposed architecture using a network simulator to validate the protocols and assess the system behavior. An outlook of the opportunities created by the architecture that call for more intense research is given in Section 6. The positioning of our work is covered in the related research Section 7.

2. 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.

2.1 Applying RDMA

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 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 where the application keeps it.

The transfer of data 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. A key concept behind RDMA is *direct data placement* which is a mechanism whereby data is enriched with local placement information such that the RNIC is able to directly access the data in main-memory using DMA.

Thanks to this RNIC, the CPU(s) of neither host are involved in the data transfer and are free to perform other tasks. The RNIC also has a TCP offload engine built in such that it can perform the network stack processing autonomously.

2.2 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) [5]. A rule of thumb in network processing states that about 1 GHz in CPU performance is necessary for every 1 Gb/s network throughput [12]. 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 1 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—

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

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

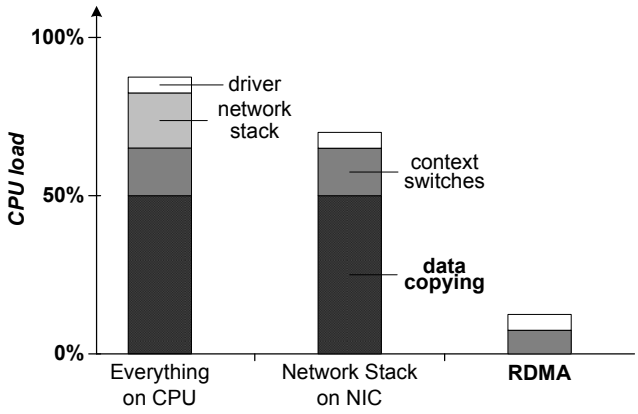


Figure 1: Only RDMA is able to significantly reduce the local I/O overhead induced at high speed data transfers [13].

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. Thus only RDMA is able to deliver 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 The 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 [15] with regard to its explicit buffer management.

The ideal scenario to benefit from RDMA is when the buffer elements have big sizes, a dozen or hundreds megabytes. Furthermore, due to the need of memory registration, normally an expensive operation, the connections between the nodes should be point to point.

Due to these costs and requirement, not every application can fully benefit from RDMA. However, the *Data Cyclotron* is an architecture that clearly can. It aims to transfer big chunks of data and it uses a ring topology, i.e., each node has a point to point connection with its neighbors.

The Data Cyclotron is built for scenarios where the workloads require huge blocks of data to be shifted from node to node at high speed. These are requirements satisfied by the latest reports on the network technology. The infiniband bandwidth will reach the 1000Gb/s in next three years. The roadmap for RDMA shows the possibility of having 12 lines of 20GB/s on RDMA cards [3].

3. BACKGROUND

The Data Cyclotron is designed from the outset to extend the functionality of an existing DBMS. For this we use MonetDB, since its inner workings are well known and expertise is close at hands. As will be clarified shortly, this does not limit the solution proposed as realization of the Data Cyclotron differs only slightly on other platforms. We briefly review MonetDB's basic building blocks, its architecture, and its execution model to make this paper self-contained focusing on the processing model and illustrated by an SQL:2003 example [4].

[3]http://www.infinibandta.org/img/technology/roadmapi_diagram.gif

```
function user.s1_2():void;
   X1 := sql.bind("sys","t","id",0);
   X6 := sql.bind("sys","c","t_id",0);
   X9 := bat.reverse(X6);
   X10 := algebra.join(X1, X9);
   X13 := algebra.markT(X10,0@0);
   X14 := bat.reverse(X13);
   X15 := algebra.join(X14, X1);
   X16 := sql.resultSet(1,1,X15);
   sql.rsCol(X16,"sys.c","t_id","int",32,0,X15);
   X22 := io.stdout();
   sql.exportResult(X22,X16);
end s1_2;
```

Table 1: Selection over two tables

3.1 Architecture

MonetDB is a modern fully functional column-store database system [7, 29, 10]. 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 lookup 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.

3.2 Query Processing

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. Table 1 illustrates the MAL plan produced for a select over two tables[5]:

```
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

[4]The system including our extensions can be downloaded from http://monetdb.cwi.nl

[5]Details on the plan and MAL optimizers can be found on http://monetdb.cwi.nl

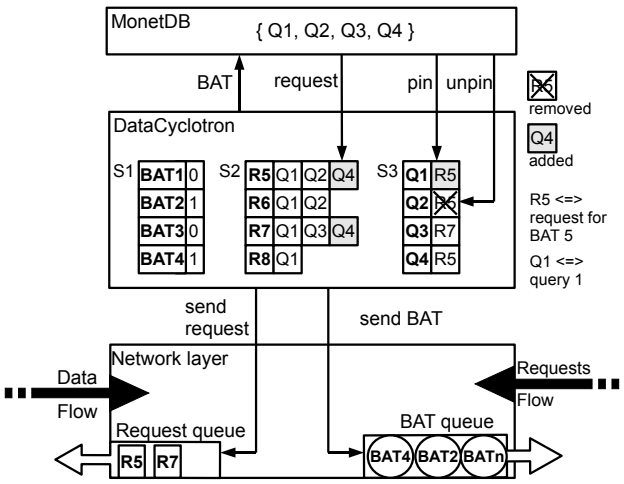


Figure 2: Internal node structure

column, reducing it using a filter expression or joining it with another column, until the results tuples are constructed. The last part constructs the query result table.

The query template is processed by a SQL-specific chain of optimizers before taking it into execution. The MAL program is interpreted in a linear fashion. The overhead of the interpreter is kept low, well below one μsec per instruction.

4. DATA CYCLOTRON ARCHITECTURE

The Data Cyclotron (DC) 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 2). The Data Cyclotron layer contains the DC runtime itself and a DC 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. For the remainder, we assume each partition to be an individual BAT easily fitting in main memory of the individual nodes. Furthermore, the BATs are randomly assigned to nodes in the ring where the local DC data loader becomes their owner and administers them in its own catalog (Structure S1 Figure 2). The BAT owner node is responsible for putting it into or pulling it out of the hot set occupying the storage ring. Infrequently used BATs are retained on a local disk at the discretion of the DC data loader.

Subsequently, queries are compiled by MonetDB into plans that call upon the Data Cyclotron layer to request or to receive a data partition, i.e., a BAT. These requests are the core information used by the Data Cyclotron to interact with the network layer for the maintenance of the hot set in the storage ring. The network layer has two queues, i.e., network buffers to encapsulate the details of passing the BAT requests and BATs around. The data is moved through the ring clockwise, i.e., a node sends BATs, stored in *BAT queue*, to its successor and it receives BATs from its predecessor. The BAT requests, stored in *request queue*, are sent anti-clockwise to reduce the latency when a requested BAT is already on its way.

The behavior of each layer and their interaction is studied in more detail in the subsequent sections.

```
function user.s1_2():void;
   X2 := datacyclotron.request("sys","t","id",0);
   X3 := datacyclotron.request("sys","c","t_id",0);
   X6 := datacyclotron.pin(X3);
   X9 := bat.reverse(X6);
   X1 := datacyclotron.pin(X2);
   X10 := algebra.join(X1, X9);
   X13 := algebra.markT(X10,0@0);
   X14 := bat.reverse(X13);
   X15 := algebra.join(X14, X1);
   X16 := sql.resultSet(1,1,X15);
   sql.rsCol(X16,"sys.c","t_id","int",32,0,X15);
   X22 := io.stdout();
   sql.exportResult(X22,X16);
   datacyclotron.unpin(X6);
   datacyclotron.unpin(X1);
end s1_2;
```

Table 2: MAL plan after DcOptimizer

4.1 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()*. The *request()* identifies the required BATs. The *pin()* and the *unpin()* mark the time when a BAT is needed and subsequently released. They exchange resource management information between the DBMS layer and the DC runtime.

The optimizer replaces each *BAT bind* call by a *request()* call and keeps a list of all outstanding BAT requests. For each relational operator argument, it checks if it comes from the Data Cyclotron layer. Its first utilization leads to injection of a *pin()* call into the plan. Likewise, the last reference of a variable is localized and an *unpin*() call is injected.

The code in Table 2 is the MAL program from Table 1 after being massaged by the DC optimizer. The MAL plan is executed using concurrent interpreter threads following the dataflow dependencies. Unlike the *pin()* call, the *request()* and *unpin()* calls do not block threads. For the *pin()* call the thread blocks until the BAT is available at the DBMS layer.

Note that such plan transformations are straightforward to integrate in a wide range of query optimizers and execution engines. Including the buffer manager of a traditional relational database engine.

4.2 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.

4.2.1 DBMS and Data Cyclotron interaction

Consider the steps taken to execute the plan in Table 2. If the BAT is owned by the local DC 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 2). 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

Request Propagation
input: the request node's origin `owner` and the requests 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 ( bat_is_loaded( &S1, reqid) )
10:        exit;
11:    if ( bat_can_be_loaded( reqid ) )
12:       if ( bat_is_already_pending(reqid) )
13:          bat_load( reqid, chk_size);
14:       untag_bat_pending(reqid);
15:       exit ;
16:    else
17:        if ( !bat_is_already_pending(reqid) )
18:           tag_bat_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);
```

Figure 3: Request Propagation Algorithm

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*.

4.2.2 Peer interaction

The *Request Propagation* algorithm, illustrated in Figure 3, handles the requests in *S2*. There are six possible outcomes for this algorithm. First outcome, if the message is received 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.

Second outcome, if the node is the BAT owner, but the BAT was already (re-) loaded into the hot set, the request is ignored. Third outcome, if the BAT was not yet loaded, but the storage ring is full the BAT is marked as *pending*, i.e., the load is postponed until hot set adjustment decisions are made. Fourth outcome, 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.

Fifth outcome, if the node is not the BAT's owner, nor the request originator, but has the same request outstanding, the request is absorbed. Otherwise, it is just forwarded.

The preferred 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 *BAT Propagation* algorithm as depicted in Figure 4.

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. For example, in Table 1, a reference for the BAT *t_id*, is passed through the variable *X1*. This memory region is freed by the *unpin()* call.

BAT Propagation
input: the bat loader's id `owner`, `bat_id`, `loi`, `copies`, `hops` ,`cycles`
output: forwards the BAT

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

Figure 4: BAT Propagation Algorithm

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

4.2.3 Storage Ring Management

The BATs in the storage ring carry an administrative header used by its owner for hot set management. The *BAT Propagation* algorithm (Fig. 4) updates the variables *hops* and *copies*. The former denotes the number of hops since it left its 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 *BAT 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.

4.3 The Network Layer

The network layer of the Data Cyclotron manages the stream of BAT messages in the storage ring and their request messages. BAT messages contain the fields *owner*, *bat_id*, *bat_size*, *loi*, *copies*, *hops*, and *cycles*. If the local node is the BAT's owner, the algorithm *hot data management* (cf., Figure 5) is called for hot set adjustments. Otherwise, it calls the *BAT Propagation* algorithm (Fig. 4). BAT request messages contain the variables, *owner* and *bat_id*. The *Request Propagation* algorithm (Fig. 3) 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.

4.4 Hot Set Management

The BATs in circulation are considered hot data. They flow as long as they are considered important for the query workload. The metric to measure this is called the level of interest, LOI for short,

New level of interest
input: the `bat_id`, `loi`, `copies`, `hops`, `cycles`
output: forwards the BAT or unloads the BAT.

```
01: /* Check if the node is the BAT loader */
02: if ( node_is_the_loader(bat_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_bat( bat_id );
09: else
10:     forward_bat(bat_id, new_loi, copies, hops, cycles);
```

Figure 5: Hot Data Set Management

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 BAT's owner when it completes a cycle. Subsequently, the new *loi* is then calculated as follows:

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

The previous LOI for a BAT 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 the multiplication of the number of copies average $CAVG$ in the last cycle by the actual *cycles* value:

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

The division by the number of cycles applies an age weight to the formula. Old BATs carry a low level of interest, unless re-newed 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 BAT is removed from circulation. If not, the LOI variable is set with the new LOI value and the BAT is sent back to the ring. This *hot data management* algorithm (Figure 5) is executed at the Data Cyclotron layer for all BATs 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 *BAT queue* load.

An overloaded ring increases the probability to postpone a BAT 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 BATs in the ring by increasing the threshold $LOIT_n$. $LOIT_n$ is stepwise increased until the pending local BATs can start moving. However, if the threshold is increased too much, the life of a BAT in the ring will reduce. This can lead to a thrashing behavior. The impact of the threshold choice is studied in more detail in section "Experiments". The model to manage the hot set might not be optimal, but it is robust and behaves as expected. Alternatives are described in an upcoming paper.

5. EXPERIMENTS

We study the Data Cyclotron behavior using NS-2[6], a popular simulator in the scientific community.

Setup. 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. No direct modifications were applied to the kernel of the simulator.

The base topology in our study is a ring composed of ten nodes. Each pair of nodes is interconnected through a duplex-link with 10 Gb/s bandwidth, 350 us delay, and DropTail as full queue policy, i.e., drop packets from the tail of the queue. Each node contains 200 MB for the *BAT queue*, i.e., network buffers, resulting in a total ring capacity of 2 GB. These characteristics comply with our RDMA-equipped compute cluster, the target for the complete system.

In our detailed analysis we use a raw data-set of 8 GB composed of 1000 BATs with sizes varying from 1 MB to 10 MB. The BATs are uniformly distributed over all nodes, giving ownership over about 0.8 GB of data per node.

The workload is restricted to queries that access remote BATs only. For, we are primarily interested in the adaptive behavior of the ring structure itself.

Experimental outline. We start our experimental evaluation using micro-benchmarks to validate the correctness of the Data Cyclotron protocols. 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.

The experimental section is completed with a real benchmark, *TPC-H*. It highlights the capability of Data Cyclotron to handle real workloads and to achieve high throughput rates.

5.1 Limited Ring Capacity

The Data Cyclotron aims to keep only the hot-data in rotation by adjusting the minimum level of interest for BATs on the move. The minimum level of interest ($LOIT_n$) is the threshold which defines if a BAT is considered hot or cold. A high $LOIT_n$ level means a short life time for the BATs in the ring, and vice verse. The right $LOIT_n$ level and its dynamic adaptation are the issues to be explored with this experiment.

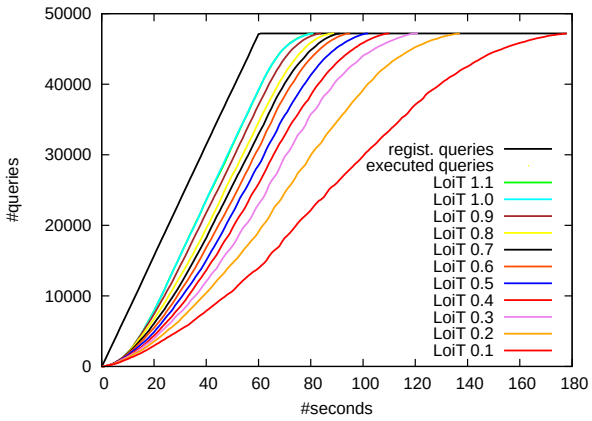
The experiment consists of firing 80 queries per second on each of the 10 nodes over a period of 60 seconds, and then letting the system run until the execution of all 48000 queries have finished. We use a synthetic workload that consists of queries requesting between one and five randomly chosen BATs.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

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 in steps of 0.1. Between two runs, the ring buffers are cleared, i.e., all the data is unloaded to the local disk.

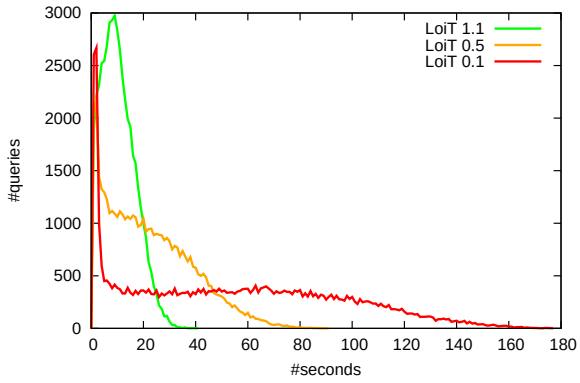
Figure 6 a) shows the Data Cyclotron throughput for each $LOIT_n$ iteration, i.e., the cumulative number of queries finished over time. The line *registered queries* represents the cumulative number of queries fired to the ring over time.

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

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

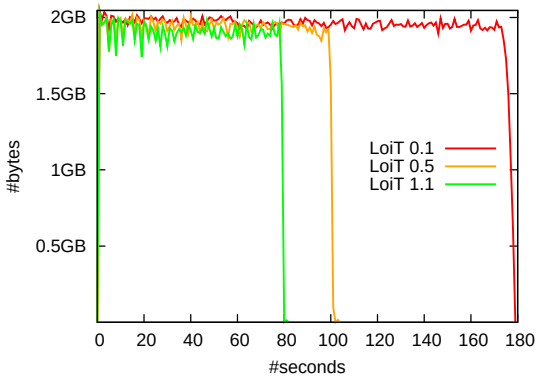


(a) Query Throughput

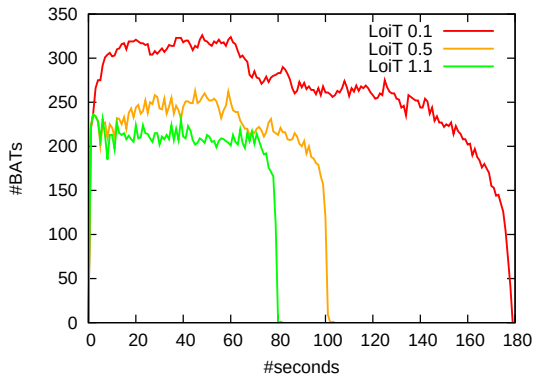


(b) Query Life time

Figure 6: Multiple $LOIT_n$ levels



(a) Load in Bytes



(b) Load in BATs

Figure 7: Ring Load

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 6 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 til its execution 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 BATs 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 BATs is delayed, i.e., the pending BATs list at each node grows. Consequently, execution of queries that wait for the pending BATs is delayed.

With optimized load for pending BATs, presented in Section 4.4, and a low drop rate, the tendency is to leave the big BATs for last. Whenever the least interesting BAT is dropped from the ring, the available slot can only by filled with a pending BAT of at most the size of the dropped one. Consequently, the ring gets loaded with more and more small BATs, decreasing the chance of big BAT loads even further. Only once there are no more pending request for small BATs, the ring slowly empties, finally making room for the pending bigger BATs. The graphs in Figure 7 a) and b) identify the BAT size trend over time. The correlation between the ring load in bytes (Fig. 7a) and the ring load in BATs (Fig. 7b), shows the BAT length in the hot set over time. With a continuously overloaded ring and a diminish in the number of BATs loaded, the graphs depict that the load of big BATs is being postponed. Therefore, the queries waiting for these BATs stay pending almost until the end. The delay gets more evident for low $LOIT_n$ levels.

This experiment confirmed our intuition that the $LOIT_n$ is inversely proportional to the local *BAT queue* load. It also proves that $LOIT_n$ should not be static. It should dynamically adapt using the local *BAT queue* load as 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.

5.2 Skewed Workloads

The Data Cyclotron is also tested for a turbulent scenario. In this experiment it is confronted with several skewed workloads *SW*, brute changes in the hot set *H*, and resource competition by the disjoint hot sets *DH*.

A skewed workload *SWi* only uses a subset of the entire database. The hot set *Hi* used by *SWi* has disjoint data *DHi* which is not used by any other skewed workload.

Each *SWi* can enter in the Data Cyclotron at different times. In some cases they meet in the system, in other cases they initialize after the end of the previous ones. These arbitrary initializations require a dynamic and fast reaction by the Data Cyclotron. If *SWj*

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 3: Workload details

enters the ring while *SWi* is still in execution, the Data Cyclotron needs to share resources between the *DHi* and *DHj*. The Data Cyclotron must remove *DHi* BATs with low LOI to load the new *DHj* data to keep the throughput high. However, the BATs from *DHi* needed to finish *SWi* queries, must remain in the ring to keep the latency low.

This dynamic and quick adaptation is studied here to answer the three major questions: How fast is the Data Cyclotron reaction to load data 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* accesses uniformly 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 BATs 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 3 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 is below the 40% of its capacity, the $LOIT_n$ is decreased one level. For this experiment, we used three levels, *0.1, 0.6, 1.1*.

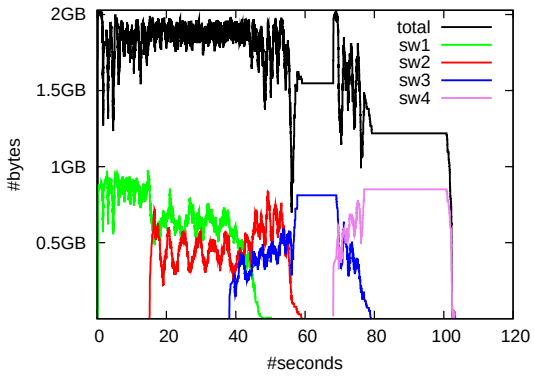
The Graph 8 a) shows the space used, in the ring, by each *DHi*. While, Graph 8 b) shows the amount of queries finished for each *DHi*.

Reactive behavior. The results show how quickly the Data Cyclotron reacts to the change of workload characteristics. The graph in Figure 8 b) shows between the 14th and 16th seconds a peak of 2000 finished *DH2* queries. The graph 8 a) in the same period shows a peak in the load of *DH2* BATs. With the initialization of *SW2* at 15 second, the peak confirms the quick reaction time. The same phenomenon is visible for all the other workloads.

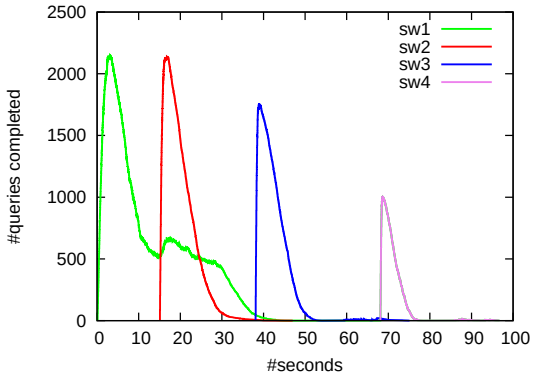
Post workload changes. The ring was loaded with data from *DH2*, however, the data from *DH1* was not completely removed. This is a consequence of the 50% time overlapping between *SW1* and *SW2*. In Graph 8 b) is visible *SW1* queries until the 43th second. The BATs to resolve these queries are kept around as it is shown in Graph 8 a). The Data Cyclotron in the presence of a new workload does not remove all the data from the previous workload until all the queries are finished. It shares the resources between both workloads as predicted. Observe that the sharing of ring resources gets lower as the time overlapping between *SW* decreases.

Exploiting the available resources. The *SW3* workload shows an interesting reaction of Data Cyclotron when it encounters a semi-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 as much space as possible. No more *SW1* and *SW2* queries exist in the system. Therefore, the last BATs for *DH1* and *DH2* start to be removed from the ring. Their removal drops the ring load down to 37,5% of its capacity,



(a) Ring Load



(b) Query Throughput

Figure 8: Skewed workload

below the 40% barrier defined for this experiment. With this load the $LOIT_n$ is set to its minimum level, i.e, the BATs are now staying longer in the ring.

With a big percentage of *DH3* queries concluded, Data Cyclotron does not remove the *DH3* BATs. 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* BATs longer, i.e, expecting they will be used in near future.

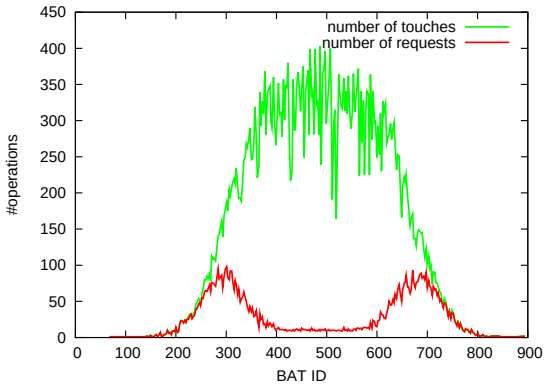
The abundance of resources is over when the *SW4* workload initializes. The ring becomes again overloaded moving the $LOIT_n$ to higher levels. Therefore, the *DH3* data starts to be removed.

5.3 Non-uniform Workloads

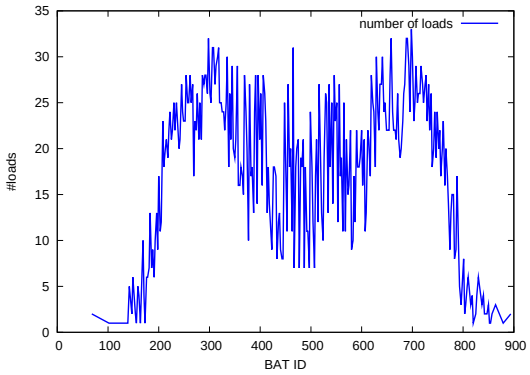
So far we have studied the Data Cyclotron architecture using uniform distributions for the BATs size and the data access. Leaving the uniform scenarios behind, we move towards workloads with different data access distributions.

In the previous experiments, the study of the BAT LOI focused on their age. The average of copies per cycle, i.e., the ring interest, was not included due to the uniform BAT access. Therefore, we initiated an experiment to stress it using a *Gaussian* data access distribution. However, the uniform distribution for BAT sizes is retained, because a uniform partition scheme can be used to break non-uniform BATs into uniform BATs.

The used scenario is the one defined in section 5.1 with exception for the data access distribution. The *Gaussian* distribution is centered around BAT id 500 with a *standard deviation* of 50. All



(a) Copies/Requests Distribution



(b) Loads Distribution

Figure 9: *Gaussian* workload

the nodes use the same distribution.

In the presence of this type of workload, the Data Cyclotron keeps the popular BATs longer in the ring and exploits the remaining ring space for the less popular BATs.

The distribution of this workload is represented as green curve in graph 9 a). The *in vogue* group is constituted by the BATs with id between 350 and 600 which were touched more than 250 times. The BATs in the border of this group, the *standard* BATs, have a lower rate of touches. The remaining ones, with less than 20 touches, are the *unpopular* BATs.

The *in vogue* BATs are highly used by the query workload, i.e., their LOI is always on high levels. Therefore, they are kept longer in the ring. Their low load rate, pictured as blue in graph 9 b) is explained by the Data Cyclotron cold down process. With a overloaded ring and the $LOIT_n$ at its highest level, the Data Cyclotron removes BATs to get room for new data. The first ones to be removed are the ones with low LOI, i.e., first the *unpopular* then the *standard* BATs. For this reason the *in vogue* are the ones staying longer periods as hot BATs.

The *standard* BATs are then requested by queries triggering their load. It is this resource management to maintain the latency in low values, that makes the *standard* BATs to be more frequently in and out of the ring.

The low rate of requests, represented as red, for the *in vogue* BATs contradicts the common believe that *in vogue* BATs should be the ones with high rate of requests, thereby high rate of loads. The reason is the requests management at 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 a *in vogue* BAT request to be *pinned* for all its queries, at once, is too low. As a consequence, the requests stays longer in the node.

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* BATs as long as possible in the ring. For a low latency, the $LOIT_n$ is used at its high level to reduce the time access to the many *standard* BATs.

5.4 TPC-H Workload

Our last experiment starts with a calibration of the simulator using traces from TPC-H ran against a single node MonetDB instance. For each of the 22 *TPC-H* queries we trace their execution for scale factor five (SF-05). Such traces contain the execution time for each operator as well as the information about intermediate result sizes.

Calibration. The data-set is composed of BATs used by each query in TPC-H. They are the columns touched by the queries and the indexes created for the TPC-H tables to speed up foreign key processing. For them, *request* calls are scheduled.

The scheduling algorithm for the *pin* calls can be exemplified using the code in Table 2. The first *pin* call, *pin(X3)*, is scheduled *OpT1 msec* after the query registration. The second one, is scheduled *OpT2 msec* after the *X3* reception by the previous *pin* call. The *OpTx* for a *pin* call is the sum of all operators execution times, since the last *pin* call, until the actual *pin* call.

A query is finished *T msec* after, the sum of the remaining operators' execution times, after the last *pin* call. The execution time of a query with *X pin* calls is the sum of all *Optx* plus *T*.

Setup. In total, the workload for each node contains 1200 queries. The query registration rate is 8 queries per second, i.e., it takes 150 seconds to register all queries.

The scheduling of the queries follows a Gaussian distribution with mean 10 and standard deviation 2. On this distribution the fastest queries are the ones with higher probability to be scheduled. With this workload distribution we stress the latency for data delivering instead of the node resources, i.e., main memory and CPU processing cost.

Each node is composed by four cores and the calls scheduling is distributed amongst them. For simplicity of our first exploration, we assume that all nodes have ample main memory to hold the intermediate results.

The scheduling at each core is done using a time line. An operator execution is scheduled at certain moment and it has a duration, *Tx msec*. A core can only be used for a single operator. The difference between the simulation duration and the sum of the operators duration defines the idle time of the core.

In theory, the execution time of a complete workload on a simulated single node and on MonetDB should be the same because all the data is local, i.e., there is no latency for remote data access. However, as shown in table 4, it is not the case under all circumstances. The reason is the optimal parallelization achieved in the simulator. The *CPU* utilization is near to optimal, 99%, while in MonetDB the threads management and the context switches between clients brings the *CPU* load to lower values (the value presented is average of the CPU loads reported by the *top* command in Linux).

To increase throughput, i.e., the number of queries per second, more nodes are added to the ring. It leads to increase latency in accessing the data. This is the only extra cost added to the queries execution time when the CPU utilization is optimal.

In table 4 the execution time is considerably increased when a second node is added to the ring. However, the throughput in-

#nodes	exec(sec)	throughput	throughP/node	CPU%
MonetDB	420	2.8	2.8	70
1	317	3.8	3.8	99.7
2	346.7	6.9	3.4	92.0
3	350.1	10.3	3.4	91.5
4	351.1	13.7	3.4	88.8
5	352.3	17.0	3.4	89.4
6	360.7	19.9	3.3	88.2
7	366.2	22.9	3.2	86.5
8	371.27	25.8	3.2	85.3

Table 4: TPC-H SF-5

creased 55%. The gain gets more evident with the addition of more nodes. From 3 to 5 nodes, the execution time has a low variance, but the system throughput increases 33% for each added node. Moreover, the throughput per node remains the same.

Addition of more nodes increases the throughput, but also has draw-backs. A diminishing return on investment becomes visible after a limited number of nodes. The throughput for a ring with six nodes or more increases 33%, but the queries execution time gets too high and the throughput per node decreases.

The *CPU* utilization shows the latency added to the system. From a optimal *CPU* utilization on a single node, the *CPU* utilization came slowly down to 15% of its capacity, i.e., 4% per core, for 8 nodes ring. Converting to seconds, each core had a total idle period of 14 seconds out of the 371.27 seconds. The same value can be derived from difference between the queries execution times of one node and the eight nodes ring.

The experiment shows that the Data Cyclotron is capable to keep the *CPU* utilization at high levels even if extra network latency is added to the global latency.

The optimal utilization is not the case for a real DBMS, as pictured by the MonetDB results. Furthermore, in [14] the simple prototype of a distributed join implementation using RDMA uncovered the limiting performance factor, i.e., main-memory speed. These points stress our conviction that ring structured storage ring and the continuous data movement is not as evil as it seems in the first place. It enables to achieve high throughput with reasonable extra latency in Data Cyclotron.

6. FUTURE OUTLOOK

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

6.1 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 BAT 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. The number of sub-queries depend 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.

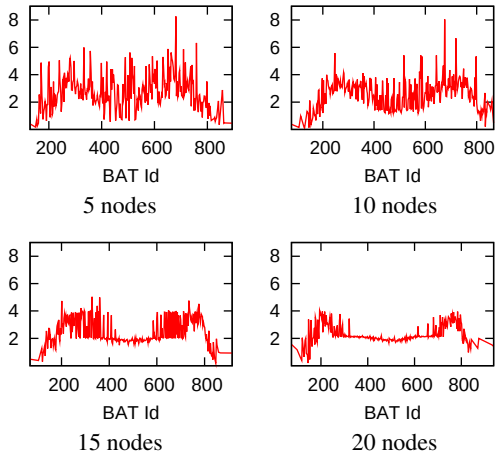


Figure 10: Maximum Request Latency

6.2 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 queries being interested. 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.

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.

6.3 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. Contrary, 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. Updates to the ring are localized to its two (envisioned) neighbors.

The decision to leave a ring can be made locally, in a self-organizing way, based on the amount of data and 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.

Extending a ring calls for a named service, where nodes are

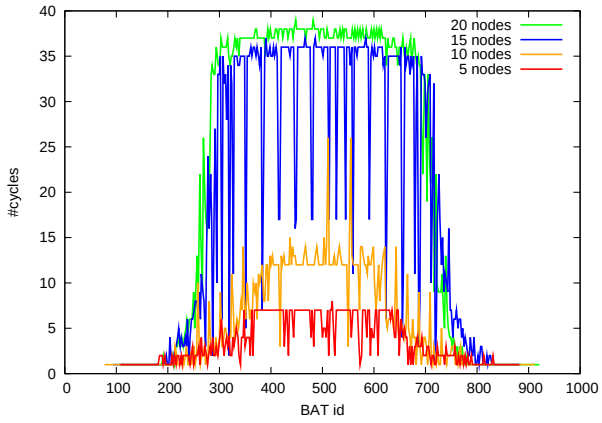


Figure 11: Maximum Number of cycles per BAT

awaiting a call of duty. When an overload occurs at one of the nodes, e.g., a strong increase of the query load, then this service is called for.

A peek- preview experiment, with the scenario defined in section 5.3, illustrates the impact on the latency when the ring grows or shrinks. The workload in the system, i.e., the total number of queries, is kept stable while the number of nodes is increased from 5 up to 20. During the experiment we observed, for every five nodes added, a latency growth of 75% in the BAT cycle duration.

The assumption that a low latency in the data access decreases requests latency is not complete correct. The experiment shows that the ring with highest number of nodes is the one with the lower maximum request latency, see Figure 10. The explanation is pictured in Figure 11, which shows the maximum number of cycles per BAT. The ring with 20 nodes, has *in vogue* BATs with more or less 38 cycles, i.e., the life of them is almost the duration of the experiment. Therefore the access cost to these BATs is only affected by the latency of its movement in the ring.

The ring with 5 nodes has a low BAT move latency, but the access cost is affected by the ring capacity. With low capacity, the *in vogue* BATs have short life (red curve in Figure 11), therefore, they are cooled down more frequently. The access cost then also includes the cost for re-loading the BATs from disk.

6.4 The Space for Updates.

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

We use a multi-version approach to support simple updates. Update requests are handled in the same fashion as queries. An update query searches for a controlling node N to settle and waits for relevant BATs to pass by. The only difference is that when a node N processes an update request, for a BAT f, it propagates f with a tag: “updating". This way, any concurrent updates, waiting in the rest of the ring, refrain from processing f, recognizing its stale state; they have to wait for the new version. Alternatively, they can be sent directly to N. Read-only queries that do not necessarily require the latest updated version can continue using the flowing old version.

The effect of updates is that the system becomes polluted with several versions of a BAT. The maintenance of these versions is as the rest of the data, i.e., using its LOI.

7. RELATED WORK

Distributed query processing to exploit remote memories and high speed networks to improve performance in OLTP has been studied in [19]. It is also addressed in the area of distributed systems as data broadcasting and multi-cast messaging, such as [18, 6, 1, 2, 4]. Solutions include scheduling techniques at the server side[1, 3], caching techniques at the client side[1], integration of push-based and pull-based broadcasts [2], and pre-fetching techniques [4]. Most systems ignore the data content and their characteristics [21]. The seminal DataCycle [18] and Broadcast Disks approach [1] are exceptional systems to be looked at more carefully.

The DataCycle [18, 6] 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 distinguish amongst the participants, all nodes access the data and contribute with data, i.e., there is no central pump. Each node bulk loads database content structured by a relational scheme and not as 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.

The Broadcast Disk [1] 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 prefetching 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. It does not combine client requests to reduce the stress on the channel.

In [2] 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. Contrary, 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 [2], 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 struc-

turing mechanism.

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*. For example, in a Distributed Hash Table (DHT), such as [25, 26], 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. CONCLUSIONS

The Data Cyclotron architecture is a response to the call-of-arms in [20], 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 we let the database hot set continuously move 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 and 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 performance consequences are studied extensively with a network simulator using the Data Cyclotron protocols and calibrated by performance traces from MonetDB on TPC-H. They confirm our intuition that a storage ring based on the hot set can achieve high throughput and low latency. Moreover, the experiments stipulate 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, and the query optimization strategies and updates all require a thorough re-evaluation in this context.

Acknowledgements

We thank the members of the Database group, in particular, Stefan Manegold for constructive advice on earlier versions of this paper. We also thank the feedback about RDMA given by Philip Frey and Jens Teubner from ETH Zurich.

9. REFERENCES

[1] Swarup Acharya, Rafael Alonso, Michael Franklin, and Stanley Zdonik. Broadcast Disks: Data Management for Asymmetric Communication Environments. In *SIGMOD '95*, pages 199–210, 1995.

[2] Swarup Acharya, Michael Franklin, and Stanley Zdonik. Balancing push and pull for data broadcast. In *SIGMOD '97*, pages 183–194, 1997.

[3] Demet Aksoy and Michael Franklin. Scheduling for large-scale on-demand data broadcasting. In *IEEE INFOCOM*, pages 651–659, 1998.

[4] Demet Aksoy, Michael J. Franklin, and Stanley B. Zdonik. Data staging for on-demand broadcast. In *VLDB '01*, pages 571–580, 2001.

[5] Pavan Balaji. Sockets vs rdma interface over 10-gigabit networks: An in-depth analysis of the memory traffic bottleneck. In *In RAIT workshop âĂŹ04*, 2004.

[6] Sujata Banerjee and Victor O. K. Li. Evaluating the distributed datacycle scheme for a high performance distributed system. *Journal of Computing and Information*, 1, 1994.

[7] P. Boncz, M. Kersten, and S. Manegold. Breaking the Memory Wall in MonetDB. *Commun. ACM*, 51(12):77–85, 2008.

[8] Sumit Kumar Bose, Srikumar Krishnamoorthy, and Nilesh Ranade. Allocating resources to parallel query plans in data grids. In *GCC '07*, pages 210–220, 2007.

[9] Surajit Chaudhuri and Vivek R. Narasayya. Self-tuning database systems: A decade of progress. In *VLDB '07*, pages 3–14, 2007.

[10] R. Cornacchia, S. Héman, M. Zukowski, A. de Vries, and P. Boncz. Flexible and efficient ir using array databases. *The VLDB Journal*, pages 151–168, 2008.

[11] D. DeWitt and J. Gray. Parallel Database Systems: The Future of High Performance Database Systems. *Communications of the ACM*, 35(6):85–98, 1992.

[12] A. Foong, T. Huff, H. Hum, J. Patwardhan, and G. Regnier. TCP performance re-visited. In *Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 70–79, 2003.

[13] P. Frey, R. Goncalves, M. Kersten, and J. Teubner. A spinning join that does not get dizzy. (under submission).

[14] P. Frey, R. Goncalves, M. Kersten, and J. Teubner. Spinning relations: High-speed networks for distributed join processing. In *DaMoN '09*, pages 27–33, 2009.

[15] Philip W. Frey and Gustavo Alonso. Minimizing the hidden cost of RDMA. In *ICDCS '09*, pages 553–560, 2009.

[16] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. Gputerasort: High performance graphics co-processor sorting for large database management. In *SIGMOD '06*, pages 325–336, 2006.

[17] Ismail Omar Hababeh, Muthu Ramachandran, and Nicholas Bowring. A high-performance computing method for data allocation in distributed database systems. *The Journal of Supercomputing*, pages 3–18, 2007.

[18] Gary Herman, K. C. Lee, and Abel Weinrib. The datacycle architecture for very high throughput database systems. *SIGMOD Rec.*, 16(3):97–103, 1987.

[19] Sotiris Ioannidis, Evangelos P. Markatos, and Julia Sevaslidou. On using network memory to improve the performance of transaction-based systems, 1998.

[20] Martin L. Kersten. The Database Architecture Jigsaw Puzzle. In *ICDE '08*, pages 3–4, 2008.

[21] Shinya Kitajima, Tsutomu Terada, Takahiro Hara, and Shojiro Nishio. Query processing methods considering the deadline of queries for database broadcasting systems. *Syst. Comput. Japan*, 38(2):21–31, 2007.

[22] Donald Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, 2000.

[23] Mong Li Lee, Masaru Kitsuregawa, Beng Chin Ooi, Kian-Lee Tan, and Anirban Mondal. Towards self-tuning data placement in parallel database systems. *SIGMOD Rec.*, 29(2):225–236, 2000.

[24] Holger Märtens, Erhard Rahm, and Thomas Stöhr. Dynamic query scheduling in parallel data warehouses. *Concurrency and Computation: Practice and Experience*, 15(11-12):1169–1190, 2003.

[25] S. Ratnasamy et al. A Scalable Content-addressable Network. In *SIGCOMM '01*, pages 161–172, 2001.

[26] I. Stoica et al. Chord: A Scalable P2P Lookup Service for Internet Applications. In *SIGCOMM '01*, pages 149–160, 2001.

[27] C. T. Yu and C. C. Chang. Distributed Query Processing. *ACM Computing*, 16(4), 1984.

[28] Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy Lohman, Adam Storm, Christian Garcia-Arellano, and Scott Fadden. Db2 design advisor: Integrated automatic physical database design. In *VLDB '04*, pages 1087–1097, 2004.

[29] M. Zukowski, S. Héman, N. Nes, and P. Boncz. Super-Scalar RAM-CPU Cache Compression. In *ICDE '06*, page 59, 2006.