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

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

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

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 6

Vision for DaCyDB

Despite the changes in the database world, relational DBMSs have tended to dominate the market. They are the default choice for holding the data behind web-applications and scientific applications. A scale-up technology which requires more CPUs, memory and IO capacity when the problem size increases. The price and size of these systems makes them un-attractive for current application scenarios with Petabyte data sets and complex had-doc queries. In response “noSQL” class of databases and MapReduce frameworks have emerged. Solutions such as Hadoop [71], Cassandra [31], Hbase [73], Membase [110], MongoDB [113], and CouchDB [43] are known for their elastic scalability, simpler data models, automatic fault tolerance, and data distribution which leads to lower administration and tuning requirements.

They are designed to scale-out rather than up. They are most appropriate for storing the types of information produced and consumed by web applications, and are optimized for access patterns and consistency versus availability trade-offs that characterize these software systems [108]. In practice, they still require some administration for performance and availability in any mission-critical data store. Most of their feature set is oriented toward the demands of web applications. Nevertheless, data in an application has value to the business and science that goes beyond the insert-read-update-delete cycle of a typical Web application. The increased complexity of data analysis is now requesting also some vertical scaling by those systems. The taken step has been the adoption of techniques and methods used from DBMSs by MapReduce frameworks, such as Clydesdale [89], or the integration with a DBMS such as HadoopDB [2].

The proposed solutions fill a hole in the solution spectrum for distribute data analysis on huge data-sets. A hole created by the decision to move from pure scale up
architectures to pure scale out architectures. It is now clear that scale-out and scale-up cannot be thought as two distinct approaches that contradict one another, but rather must be viewed as two complementing paradigms. It does not make sense to go to either of the extreme scenarios, but rather to combine them. The question then becomes, at what point should the services run locally and what time should they be distributed to achieve the scale-out model?

In this Chapter we propose the vision for DaCyDB to answer this question. The DaCyDB detaches itself from those architectures which were only designed to scale out or architectures that were only designed to scale up. It combines both approaches by exploiting the decentralized Data Cyclotron architecture and partitioned parallelism used by MonetDB (cf. Section 5.2.3). Combined with the network road-map it provides a high throughput and flexibility for turbulent workloads with optimal resource utilization.

6.1 Outline

The Chapter is organized as follows. Section 6.2 describes the DaCyDB scale out approach for distributed parallelism. Section 6.3 explains the re-use of intermediates, created by the distributed parallelism, to boost multi-query performance. Distributed parallelism for complex queries is discussed in Section 6.4. Section 6.5 describes how nodes dynamically allocate resources for the query execution, followed by Section 6.6 to propose a multi-ring architecture and fault-tolerance in Section 6.7. The Chapter ends with a summary in Section 6.10 to revive the most relevant features of DaCyDB.

6.2 Scale out approach for DaCyDB

The DaCyDB provides a seamless transition between a scale up solution to a scale out solution by having both in harmony. Based on the resources of each node the computation on-the-fly has the flexibility to explore both models in different degrees. The scale-up model is used for balanced resource utilization and explore data locality using the nodes cache. The scale-out model is used to spread the execution of complex queries across the cluster. With heterogeneous hardware, both models complement each other to achieve high throughput and optimal resource utilization.

The flexibility to move between both extremes and find the optimal niche for efficiency is given by the decentralized architecture of Data Cyclotron and the operator-at-the-time paradigm used by MonetDB. The flexibility comes from the ability to split a query plan into independent sub-query plans and each of them into multiple inde-
SELECT c.t_id FROM t, c WHERE c.t_id = t.id;

Figure 6.1: SQL statement.

The statements are executed independently. One or more sub-query plans are executed in the same node while others are executed on other nodes. Their allocation depends on the local resources of each node such as number of cores and available main memory for intermediates.

6.2.1 Plan definition

Leaders of the database community have argued [136, 35] that an overwhelming majority of structured data repositories are either star or snowflake schema. Star-schema have proved to be a good way to model large data sets in many industries. Hence, the plan definition to scale out follows a rule of thumb for distributed query parallelism. The dimension data should be replicated while fact data partitions should be spread among the nodes. It exploits data locality for a range of operators such as join. The same approach has been taken in MapReduce context to reduce the amount of data being continuously shuffled [89]. The authors have shown the performance gains for star schema workloads when the rule is applied. However, the data distribution and the query plan optimization were too complex and dependent on how the data is partitioned and assigned to the nodes.

Contrary, DaCyDB decomposes a query plan into a set of sub-functions. The sub-functions are created around the fact columns partitions. DaCyDB identifies the fact columns based on their size or information collect at loading time. The information is used by an optimizer, opt_split, to generate each sub-function. The optimizer is called after the opt_datacyclotron optimizer to have the necessary calls to interact with the Data Cyclotron.

The optimizer with the list of fact columns takes one fact partition $P$ and builds a sub-function around the instructions that use $P$ as input, or an input derived from $P$, and all their dependencies until the top of the plan. To exemplify the steps, we use the unroll MAL plan from Section 5.2.3, now pictured in Figure 6.2a. It is the MAL plan for the SQL query represented in Figure 6.1.

Assuming that table $t$ is a fact table and taking the plan in Figure 6.2a, the first function is built for column $id$ partition one ($id.1$). The search starts at the request() call and continues until it reaches an instruction which has as input another fact column, or an intermediate derived from a fact column. Such instructions are designated as
merge operators. They are often a join operator or a pack operator. The later packs sub-functions results and is represented as instruction A) in Figure 6.2a.

All instructions, except the merge instruction, are packed into a sub-function. The
sub-function built for id.1 (Figure 6.2b) contains the highlighted instructions from Figure 6.2a and a dacy.export result at the end. The dacy.export is used to publish the function plan (FP) and the result into the ring as an ordinary data chunk, called mid-result \(^1\). For query Q1 the result is column X33, see instruction B) in Figure 6.2b.

The merge operator is then moved into another sub-function (query Q) proceeded by one or more resume() calls depending on the number of dependencies on other sub-functions, see Figure 6.2c. A resume() call collects a mid-result from the storage ring. It uses FP to identify the mid-result. Once picked it resumes the query execution. The injection of resume calls interconnects the sub-functions. Any of them becomes eligible for execution once the respective mid-result becomes available in a local buffer.

The process is repeated for all fact columns partitions until the optimizer reaches the instructions to export the final result to the user. In this example, two more sub-functions are created, one for id.2 and another for id.3 \(^2\). With only two levels of sub-functions, sql.exportResult(), instruction C) in Figure 6.2b is also added to sub-function query Q, Figure 6.2c. Hence, query Q is the sub-function responsible to merge all results from sub-functions query Q1, query Q2, and query Q3, and return the final result to the user.

Each sub-function is posted on the ring for execution and the export of the final result is executed at the local node. Depending on the number of slots ate each node, one or more sub-functions are picked for execution by other nodes. Once picked, the resume instructions, together with the request instructions, are the first ones to be executed.

At the first glance, with the query broken into sub-functions, the plan seems to create some data redundancy and extra load in the storage ring. However, since the plan distribution is based on the fact partition, the largest partition, only one node requests its load which combined with caching policies, explained in Section 4.5, the large data chunk is quickly removed from circulation.

Furthermore, only the dimension data, which are orders of magnitude smaller, and mid-results, which are often a result of a selection, a filter or a join, are flowing around. Similar traffic is created in MapReduce frameworks through distributed cache \([70]\) to replicate the dimension data (cf., Section 3.6).

"Using a distributed file system has also another drawback. Sometimes all or many of the tasks in a MapReduce job list need to access a single file or a set of files. For example, when joining a large file with a small file the predominant approach is

\(^1\) The name mid-result is used to distinguish this type of intermediate results from the ones produced within a sub-function execution

\(^2\) Since their plans are very similar, their MAL plans are pictured in a later Section 6.3.3 for another discussion. In case of interest, Figure 6.8a and Figure 6.8b
to open the small file as a side file. The small file is opened directly in the map task rather than be specified as an input to the MapReduce job. It is opened, loaded into memory and used for the join in the map phase [59]. When thousands of map or reduce tasks attempt to open the same HDFS file simultaneously a large strain occurs on the NameNode and the DataNodes storing that file [59]. To avoid this situation MapReduce provides a distributed cache [70]. The distributed cache allows users to specify any HDFS file they want every task to have access to. These files are copied into the local disk of the task nodes. The approach adds extra overhead before the job can be initialized and the data might need to be re-shuffled again when different jobs are submitted.

The overhead imposed by distributed cache does not exist in DaCyDB. The dimension data is loaded into the hot-set as sub-functions are requesting it. Once loaded it is shared among all nodes, therefore, there is no need to re-shuffled the dimension data in case new jobs are assigned to different nodes.

In Section 6.3.3 we proposed a solution to reduce the load of dimension data by packing the common instruction between each sub-function into a single sub-function and re-use its output to avoid the re-load of all dimension data for future computations.

### 6.2.2 Dynamic plans

The sub-functions, interconnected through the `resume()` instructions, form a logical computation graph that is automatically mapped onto physical resources at runtime. There may be many more vertices in the graph than execution cores in the computing cluster. Such difference has been used in the MapReduce world to create waves, i.e., more map tasks than map slots. Having more sub-functions than the number of slots helps to balance the workload across the cluster.

The resources available at execution time are not generally known at the query definition time. With heterogeneous clusters, the node where the query is compiled into a set of sub-functions might have less resources, such as available main-memory, than the nodes where most of the sub-functions are executed. In this situation, the plan unroll before the sub-functions definition tends to be more conservative. It leads to inefficient resource utilization and constraining vertical scalability.

On the other hand, if the sub-functions were defined in a wealthy node, their execution will request nodes with high number of available cores to quickly process the instructions and consume the produced intermediates to not overload of the main memory with intermediates and consequently their split to the virtual-memory.

Hence, after being picked for execution, a sub-function can be unrolled or rolled up based on the local available resources. For unrolling, the rules are the same as the ones
begin query.Q1();
  X2 := request("t","id",1);
  X5 := request("c","tid",1);
  X11 := request("c","lid",2);
  X12 := request("c","lid",3);
  X24 := pin(X2);
  X17 := pin(X5);
  X18 := reverse(X17);
  unpin(X5);
  X27 := join(X24, X18);
  X30 := reverse(X27);
  X35 := leftjoin(X30, X24);
  X19 := pin(X11);
  X20 := reverse(X19);
  unpin(X11);
  X28 := join(X24, X20);
  X31 := reverse(X28);
  X36 := leftjoin(X31, X24);
  X21 := pin(X12);
  X22 := reverse(X21);
  unpin(X12);
  X29 := join(X24, X18);
  X32 := reverse(X29);
  X37 := leftjoin(X32, X24);
  unpin(X2);
  X33 := pack(X35,X36,X37);
  dacyExport("query.Q1",X33);
end query;

Figure 6.3: Unfold sub-computation query.Q1.

used in Section 5.2.3. For example, query.Q1' in Figure 6.3 is the result of unrolling query.Q1 from Figure 6.2b. Instead of packing and do a single join (instruction D and E in Figure 6.2b), the join is unrolled and the packing only occurs at the end of the sub-function. The end result is represented by independent instruction blocks in Figure 6.3.

The DaCyDB tries to extract parallelism within a query by exploiting the fact that dependencies are all explicitly encoded in the flow graph. Depending on the data distribution and workload, the number of partitions for the fact data and dimension data can be increased using dynamic slicing. Hence, through dynamic partitioning inter- and intra- sub-function parallelism is increased even further. Horizontal partitioning on the
fact data increases the number of independent sub-functions, while on dimension data increases the number of independent instructions within the sub-functions.

With dynamic plans DaCyDB has balanced resource utilization, but it also solves a resource allocation problem common in distributed processing, data skewness. For example, a sub-function $B$ is not eligible for execution until all its input arguments have been produced. Hence, if one of the inputs is from a sub-function $A$ for which the execution is been slowed down due to data skewness, $A$ is decomposed into a set of sub-functions.

The set of sub-functions is sent to ring to be picked up for processing. The merge of their outputs is then passed to $B$. In case the merge result is too big, the same approach can be applied to $B$. With the data chunk in memory, and with RDMA, slicing is a cheap procedure. To materialize it, the memory region which defines the view is sent to the storage ring to be picked for execution or stored as persistent data.

The data once re-organized is used by other queries to boost performance. The same goal has been persecuted by MapReduce frameworks to avoid skewness and have in-memory mappers. Small partitions are defined before job initialization which then results in thousand of mappers being scheduled [72]. The static definition at the job initialization time combined with their static data partition and data distribution does not always improve performance due to un-optimal resource utilization.

### 6.2.3 DaCyDB user interaction

The user expresses his/her computation through Jason, SQL, Xquery, SparQL, and SciQL. The computation is then compiled into a MAL plan. Using the procedural language from the middle layer of MonetDB, DaCyDB keeps the definition of plans and their dynamic adjustment for efficient parallelism transparent for the user.

A solution that exploits the split between the logical schema and the storage schema. It allows the user to use high level languages to express the computation and the same time access heterogeneous data sources to be processed in heterogeneous hardware. A goal persecuted since the downs of the DBMS, (cf., Section 1.3).

"In the earliest DBMS, beyond efficiency, the aim was to make the data independent of the logic of application programs, i.e., the split of the logical schema from the storage schema, so the same data could be made available to different applications."

With this clear split between the schemes and the possibility to crumble MAL plans several times into independent sub-functions (cf., Section 6.2.2), the flexibility to scale and explore all available resources makes DaCyDB to stand out from MapReduce solutions. They embed computations in a scripting language in order to execute programs.
that require more than one reduction or sorting stage. In the past years, new query processing systems were built on top of Hadoop providing different languages. The most well known are Jaql [64], Pig [123], and Hive [138]. They are used to express complex computations on the different stages of data processing.

Pig [123] and Jaql [64] are high-level languages for composing and executing complex dataflows on Hadoop. Pig and Jaql focus on supporting ETL-like workflows that require complex data transformations. Like Hive, they also support sort-merge and hash join strategies, and are also constrained to joining two tables at same time.

**Pig.**

Authors in [60] have proposed Pig. It is a dataflow system that aims at a sweet point between SQL and MapReduce. It offers SQL-style high-level data manipulation constructs which can be assembled in an explicit dataflow and interleaved with custom Map- and Reduce-style functions or executables. Pig programs encode explicit dataflow graphs as opposed to implicit dataflow in SQL.

Pig dataflows can interleave built-in relational-style operations like filter and join, with user-provided executables that perform custom processing. A schema for the relational-style operations can be supplied at the last minute. It is convenient when working with temporary data for which system-managed metadata is more of a burden than a benefit [60]. For data used exclusively in non-relational operations the schema does not need to be described at all [123].

Pig compiles these dataflow programs, which are written in a language called *Pig Latin* [117], into sets of Hadoop MapReduce jobs, and coordinates their execution. The steps to create the Hadoop MapReduce jobs are pictured in Figure 6.4. From the Pig Latin script a logical plan is created with the dependencies. Out of the logical plan a physical plan is created based on the data partitions and their location. Once created, the physical plan is broken into map- and reduce-task. The synchronization points of a physical plan are the delimiters of each task. For example, a *Global Rearrange* requires a reduce task.

**Hive.**

Hive [139] provides a relational model for Hadoop and a SQL interface called HiveQL. Hive’s tables are analogous to tables in relational databases. Each table has a corresponding HDFS directory, but it also supports external tables defined on data stored, NFS or local directories. Users can load data from external sources and insert query results into Hive tables via the load and insert data manipulation (DML) statements respectively. Multi-table insert is also supported.
Figure 6.4: Steps to create the Hadoop MapReduce jobs in Pig [60].

Through HiveQL, the user can use statements like select, project, join, aggregate, union all, and sub-queries in the from clause. Multiple queries can be performed on the same input data using a single HiveQL statement. Hive optimizes these queries by sharing the scan of the input data. It exploits cooperative work as Data Cyclotron.

Hive turns HiveQL queries into a series of independent MapReduce jobs to be executed on a Hadoop cluster. The initialization of each job and checkpoints storage between them has a considerable overhead which taxes the query response time. Solutions have emerged to reduce the overhead. The study in [116] focused on grouping MapReduce jobs that perform common computations and evaluating each group as a single job. In MapReduce Online [40] the authors modified the Hadoop architecture to allow pipeline of the intermediate data between operators. For example, the reduce output from job $A$ can be used as input for the mapper job $B$, this is, incremental input pulling.

Pig and Hive are the first step towards to the complete split between the logical schema and storage schema in MapReduce frameworks. There are also prototypes,
such as FlumeJava [32], with the same intention, but not so successful as Pig and Hive. Nevertheless, the user is confined to a single language, Latin Pig or HiveQL, to define computations. For complex queries, since they do not support all relational operators, they still rely on user-defined functions written in Java. They are seen as black boxes during query plan optimization.

6.2.4 DaCyDB data access and distribution

Depending on the type of the source nodes, the Data Cyclotron provides two data distribution models: *iterative loading* and *a priory loading* (cf., Section 3.6).

"For workloads which request data from a data source which is always online and has a stable connection with one or more nodes in the ring, the iterative mode should be the preferred one. Otherwise, all data should be loaded before any computation starts. For both, a schema is first shared among all nodes. A request is defined based on this schema and might have as target different data sources. It is composed of the source file identification, parse expression, and split function. For example, a request to extract a relational column from a CSV file, or to extract records from a data block stored in a cloud storage file."

The latter is the used one in most of the architectures for distributed processing. For DaCyDB, thanks to the Data Cyclotron flexibility and the MonetDB storage model, both are supported. It leads to an effortless data partition/distribution scheme and efficient data access.

For data partitioning and schema distribution, the Data Cyclotron does not impose any requirement other than that each data chunk fits into a single DaCy buffer. Hence, the database is freely partitioned vertically and horizontally using any database partitioning technique to increase locality of access.

**Iterative loading for relational schema.**

With the flexibility to partition in both dimensions, the *iterative loading* in the context of relation databases is achievable through a small extension to the *COPY FROM* SQL statement and an a priory load of the relational schema.

Preceding the first query execution, a user needs to specify the schema and how the tables should be populated. For *iterative loading*, the loading statements are only used to identify the columns and how to populate them, but not to load the data. Such loading information can always be updated by executing a new *COPY FROM* statement.
Figure 6.5: SQL statements to populate table T1.

Figure 6.5 has the statements to populate a four column table with data from two CSV files. Statement a) can be executed at any node and it leads to the creation of table T1 on the local schema. The statement is then propagated counter clockwise for T1 be added to all node’s schema. For each column an entry is added to the DaCy catalog and marked as unloaded.

Statements b) and c) provide the information on how column a, b, c, and d should be populated. With b), column a, b, and c is populated with data from the first, fourth, and twenty first property from t1.csv. With c), column d is populated with data from a different file, the fourth property from time.csv. The execution of b) and c) does not load any data into the DaCyDB. It only adds the loading information to the DaCy catalog. The load of the data only occurs when the first query is executed such as statement d).

The execution of statement d) triggers the load of column a and d. During the compilation of d), opt.datacyclotron inspects the DaCy catalog and if columns are marked as unloaded, it injects a load() call per each input source, so the file is scanned only once. After the load() calls, it injects a eval() call with the query as argument, see Figure 6.6.

At execution time, the load() call triggers the load of requests to load data for each column. A request contains as arguments the path to the input file Path, delimiters Ds, tokens ids Tids, number of records R, and size of the largest static column type length Clen. All this information was provided by the schema and COPY FROM SQL statements.

---

3 Other type of files are also supported.
4 For table T1, column d has the largest type, i.e., type double, because column e will only be composed of OIDs (eight bytes length) to access a string dictionary which is propagated in an independent data chunk.
begin query();
X1 := load(T1, a);
X2 := load(T1, d);
X3 := eval(X1, X2, "SELECT a, d FROM T1 WHERE d > 10.0 LIMIT 100;"); 
end query;

Figure 6.6: MAL plan for iterative loading.

A request flows counter clockwise and to be picked by a node connected to a data loader service (cf., Section 3.6).

" Independently of the data source, the Data Cyclotron provides a service to be installed at the data source called data loader. A data loader establishes a connection with the Data Cyclotron and waits for data load requests. Using different readers, the data loader extracts relevant properties or fields, partition them, and send them as data chunks to the ring. "

At its reception, the data loader, using Clen and R estimates how many records it can store in a DaCy buffer \(^5\). Then it starts reading records from Path, it uses Ds to parse each record and extract the tokens identified by Tids. At the completion of the first partition for each column, the data loader, tags them as COLUMN_LOAD, adds the estimate number of partitions, and loads them into the ring.

The partitions do a full cycle updating the DaCy catalogs in all nodes. When the first partition of each of the columns have reached the requesting node, the load() call returns and the eval() instruction is executed. Its execution evaluates the query by compiling it and sending it through the optimizers pipeline. At this time, the DaCy catalog indicates the columns are loaded, or are being loaded, and based on the number of partitions, the opt_datacyclotron injects the necessary calls to interact with DaCy layer and to exploit efficient parallelism.

**Heterogeneous data sources.**

To have DaCyDB more competitive for data preparation, the information to parse the data source, identify the data structure, and define which records or tokens are relevant, should be provided by the user. The approach is one of the advantages in using

\(^5\)By using the largest column type, the loader makes sure the OIDs inside each partition are aligned. A rule of thumb in column stores to have efficient joins instead of Cartesian products.
MapReduce frameworks, i.e., the user through inputReaders specifies how the data should be retrieved and passed to the mappers for processing. With the map function being also defined by the user, the data structure used for processing is defined by the user.

Through a simplification of the COPY FROM, DaCyDB asks the users to define a parser as SQL function which calls a C, or C++ user define function (UDF). This parser is then used by the data loader to extract data at the source nodes. However, in DaCyDB the output of such functions is always tuples to be loaded into data chunks, this is, column partitions.

For the most common data source types, e.g., CSV, FITS ⁶, NETCDF ⁷, and MSEED ⁸, the DaCyDB provides efficient parser implementations. For CSV files, DaCyDB has a data parser called csvParser. Its signature is presented in Figure 6.7.

Using the record delimiter and token delimiter, it extracts the relevant records and from them, the tokens to be sent to DaCyDB. If a sampling function is not provided, all records between the first_record and the last_record are considered relevant. Otherwise, the function is used to identify the relevant records between the first_record and the last_record.

The parser function is loaded into the system through a COPY FROM statement and sent to the data loader once the data is requested. Using the csvParser function, statement d) in Figure 6.5 is now replaced by statement e) in Figure 6.7. In this example, all records are loaded ⁹.

---

⁶http://en.wikipedia.org/wiki/FITS  
⁷http://www.unidata.ucar.edu/software/netcdf/  
⁸http://www.iris.washington.edu/manuals/SEED_appG.htm  
⁹To load all records last_record is defined as -1.
With this approach, the `COPY FROM` statement adopts a simpler syntax by replacing the `USING DELIMITER`, `OFFSET`, and `RECORDS` clauses by a SQL function. Such modification on the standard SQL is straightforward and gives a new degree of flexibility and control to the user to load and structure data from heterogeneous data sources. An important feature to drop one of the most important DDBMS drawbacks compared to MapReduce frameworks, this is, flexible data loading.

**An intelligent data loader.**

The extensions proposed for SQL `COPY FROM` statement allows the user to define iterative loads. The next step to reduce the cumulative cost in answering the first query is to efficiently extract data from the source nodes. Such efficiency is provided by the data loader. An independent service which its study and conceptualization is out of the content of this dissertation. Nevertheless, we lay down here the most important features to be considered on its conceptualization in the near future.

The optimal data loader is the one that after several iterative data load requests has only scanned the entire data source once. Since in most of the cases the record location is not know a priory, the data loader with each data extraction has to learn about the records location to avoid several scans to the same file. For example, if only few columns are extracted from a CSV file, offsets for the remain ones should be saved to extract them quickly in the next request, or even save the remain ones into separate file.

The approach has been taken by NoDB [9]. NoDB for each query retrieves the data directly from the data sources. In each extraction saves the information relative to the location of the records and tuples. Such information is used to do relative jumps in the data source to located other records or tuples, and thus speed the data extraction and reduce the cumulative cost in answering the queries. Over time, a full map over the data set is created and the queries access the data as if it has been loaded a priory into the database.

The approach is novel, however, its fine granularity creates a complex index over the data source. For huge data sets its management becomes unfeasible and too complex. The solution is designed for a central system and not as an independent service for data extraction to feed distributed processing nodes. Nevertheless, DaCyDB and NoDB share some concepts.

The DaCyDB data loader service is similar to a data service provided in cloud environments such as Azure SQL or Azure Blob 10. They provide efficient stream of data to the application. Azure SQL, allows the application to push some simple selections

---

to filter and specify which data is relevant for loading, and thus reduce network traffic from the cloud to the computational nodes. For Azure Blob, the application through a simple API, i.e., GETs and PUTs, requests data for processing. A more efficient API has been tested with a prototype. The prototype, with similar characteristics to the DaCyDB data loader, was designed to extract data into column partitions and provide efficient data access to a MapReduce framework.\footnote{The prototype consisted in an efficient layer to iteratively retrieve data from the cloud service and adaptively re-organize it to reduce data access latency. It was designed and implemented by the author of this Dissertation during an internship at Extreme Computing Group from Microsoft Research Redmond. The design was evaluated as a promising solution, which validates the research directions for DaCyDB.}

**Integrity check.**

The *iterative load* reduces the cumulative cost to answer the first query for extreme large data-sets for which only few properties are used for query processing. During the iterative loading, due to the absence of data location, the DaCyDB can also check values allowed in columns and enforce integrity between columns and tables in a distributed manner.

The simplest and most common column are the *NULL* and *NOT NULL*. If not null, type integrity is checked using the *CHECK* constraint on a column. Another common constraint is *UNIQUE*, which ensures a value in a column is unique within the table which can also be done with a *PRIMARY KEY* constraint.

When a partition enters the ring, the first nodes are responsible to do integrity checks with a single scan of the partition. For the *FOREIGN KEY* constraint, a concept at the heart of a relational database, the validation is bit more complex. The *FOREIGN KEY* relationship with values in another table. Hence, it might trigger the load of extra data into the ring, the *FOREIGN* columns. They have to be all loaded and distributed among the nodes. With the primary key columns distributed among the nodes the entire foreign column(s) is scanned giving a complete turn around the ring. With a single spin around the ring *FOREIGN KEY* validation is completed.

Not all workloads require constraint validation, but for workloads like TPC-H it is crucial for data consistency. The DaCyDB supports integrity rules which are often neglected for distributed query processing due to its high weight in the cumulative cost from the load of the first tuples until the output of the first result. DaCyDB gives the flexibility to the user to define different levels of consistency depending on the type of workload, or search intentions. For example, for a first approach a scientist might just want a raw view over the data to see if there is any interesting correlation, if yes, he can then invest in consistency for more accurate results. A vision shared by the authors of an awarded vision paper [93].
6.3 Result caching

The mid-results can be cached and re-used by other queries/jobs to avoid (part of) the processing cost, and thus boost performance. They are simply treated as persistent data and pushed into the storage ring for interested queries. Combined with the intra-query parallelism, multiple sub-functions originated from a single query in execution create a large flow of mid-results to boost others.

Like base data, mid-results are characterized by their age and their popularity on the ring. They only keep flowing as long as there is interest. To not overload the ring, mid-results are kept on the local cache of their creator. If a request is issued, they enter the ring. Otherwise, they are gradually removed from the cache to make room for new ones. The first ones to be evicted are the unpopular ones followed by the ones with low percentage of filtered data, i.e., their size is near the sub-function input size.

The idea was tested for a centralized system, Recycler [84, 83]. The Recycler speeds up query streams in a self-organizing way by exploiting an operator-at-a-time execution paradigm where complete intermediates are a by-product of every step in the query execution plan. To improve the response time and throughput in the operator-at-time setting, only the management of intermediates cost needs to be kept under control, because the creation cost is always taken by the execution paradigm.

The hypothesis was tested in the context of the operator-at-a-time database system MonetDB, the same DBMS used for DaCyDB. Its abstract machine interpreter was hooked up with a recycler optimizer and run-time module. The optimizer marks operations of interest for harvesting. The run-time support uses this advice to manage a pool of partial results. It avoids re-computation of common sub-queries by extracting readily available results from the pool. To manage the resource pool they defined their policies in three dimensions: instruction matching, investment cost versus savings, and pool administration maintenance [84, 83].

For each instruction to be executed the recycler performs a matching process, i.e., it searches for a possible reusable relational algebra operation in the recycle pool. For each operation executed the recycler decides if it is beneficial to keep the result. Finally, to prevent the pool of intermediates becoming a resource bottleneck itself, operations with low potential for reuse should be cleaned from the pool to reduce the memory usage and the search time. Traditional approaches, such as LRU, and cost-based policies based on plan semantics were used as eviction policies to select instructions for eviction. All policies respect and exploit the semantic relationships among the operations executed.

Despite recycler and result-caching have the same goal, this is, re-use operators/sub-function results to boost multi-query performance, they differ in few aspects. The recycler has finer granularity re-utilization compared to DaCyDB result-caching.
recycler is at operator level while DaCyDB does it at sub-function level. The matching on DaCyDB uses the sub-function plan and its input arguments. The match has to be exact since matching a sub-function plan is more complex than matching operators. Furthermore, using the storage ring as recycler cache provides DaCyDB the opportunity to explore remote memory to keep mid-results even in the presence of workload with high demands of main memory. In recycler it would stress the pool of partial results to be shrunk by dropping instructions with high potential for reuse.

6.3.1 Execution model

At execution time, for a sub-function \((C)\), the DaCyDB sends a request counter clockwise to determine if \(C\) was already executed or not. The request contains the sub-function plan and its input arguments identification. If the request returns marked \textit{executed}, or \textit{in-execution}, or in the mean time a matching \textit{mid-result} passes by, the \(C\) execution is dropped. Any \textit{sub-function} dependent on \(C\)'s output will collect the \textit{mid-result} from the storage ring.

The sub-function execution cancellation is only possible for the sub-functions which are not returning the result to the user. The cancellation is achieved through the registration of a request for the \texttt{dacy.export} input. At the reception of a matching \textit{mid-result}, the \texttt{dacy.export} becomes eligible for execution despite all other instructions have not yet been executed. For example, in \texttt{query.Q2'} (Figure 6.9) the input for the \texttt{dacy.export}, \(X35\), can come from instruction \(F\) or from the storage ring. Since it is the last instruction, the query can exit.

The \texttt{dacy.export} execution sets the sub-function status to \texttt{executed} so no more instructions are picked for execution by other execution threads. The last thread to exit the sub-function execution requests a garbage collection to clean context and intermediates.

6.3.2 Mid-results cache

For workloads with skewed interest and queries with several sub-functions in common, the hot-set is mainly composed of \textit{mid-results}. The number of sub-functions sent for execution is low, therefore, \texttt{popular} raw data is not requested for processing. On the other hand, for uniform workloads with high percentage of disjoint sub-functions, popular data chunks and \textit{mid-results} have to compete for the storage ring resources. In this situation, the input and output data (\textit{mid-result}) of a sub-function have high probability to co-exist in the ring at the same time.

Keeping both might improve the performance of some queries, but it might also slow down others since more data has to be forwarded in the ring. On the other hand,
removing the mid-results from circulation just after they have been re-used reduces
the chance to boost performance of future queries with expensive sub-functions due to
their computational complexity and/or large input data volumes.

To optimize storage ring utilization and have high global throughput, the hot-set
management routine needs an extra condition than the LOIT to unload mid-results in
case the input data to create them is also flowing in the ring. The condition determines
the average time to process the computation or re-use the mid-result. For determination,
the condition considers the LOI of each of the N input data chunks \(LOI_i\), the
size of each of them \(CHK_i\), the mid-result \(LOI\) and its size \(MID\), the DaCy-
cycle \(^{12}\) duration \(DCC\), and an estimation of the sub-function’s processing time \(T\),
i.e., processing time without data access latency.

For the mid-result there is not execution time. Hence, the average processing time
\(MIDACC\) is, in worst case scenario, the time to retrieve from the storage ring, i.e.,
a DaCy-cycle. For the computation, if input data is composed of more than one data
chunk, the average time to access them is half of a DaCy-cycle (cf., Equation 6.2)
because the access to one data chunk partly overlaps with the access to the others. In
both cases, the multiplication with \(LOI\) (cf., Equation 6.1 and 6.2) gives a weight
equal to popularity of each data chunk among all nodes.

For workloads with computations with almost no processing time, i.e., only data
access latency, the relative size difference between the input data and the mid-result
(cf., Equation 6.3) favors the lightest one. Selecting the lightest data chunks more
room is left on the storage ring for other data chunks or mid-results.

\(^{12}\)A DaCy-cycle is the average time for a package to be sent and return back to the origin (cf., Section 3.4.4).
Algorithm 1 Condition to check if a mid-result should be evicted from the storage ring.

\[
\text{if } \text{MIDACC} < (\text{INACC} + \text{EXEC}) \text{ then}
\]
\[
\text{drop} \leftarrow \text{False}
\]
\[
\text{else}
\]
\[
\text{drop} \leftarrow \text{True}
\]

Algorithm 1 checks if the average time to access input data plus processing time is higher than the average time to access the mid-result. If yes, the mid-result is kept in circulation, otherwise, cached. If no further request triggers its load, the mid-result is evicted from memory or stored in the cold-set. The cold-set is only reached if it has been an in-vogue mid-result. Once cold down, it is seen as a materialized view which was made persistent. Hence, result-caching, based on the workload, dynamically identifies materialized views which otherwise could only be detected by DBA or static analyzes of the query patterns.

6.3.3 Enhance result caching

Result caching can be used to solve the dimension data redundancy issue when the query plan is split into sub-functions (cf., Section 6.2.1). The sub-functions created have instructions in common. In Figure 6.8 are the other two sub-functions from \textit{query}_Q (cf., Section 6.2.1) and the common instructions between sub-function \textit{query}_Q2 and \textit{query}_Q3 are highlighted.

Since they are encapsulated into a sub-function, the result of their execution is not re-utilized by others sub-functions. Hence, for \textit{query}_Q, this highlighted code is executed three times, i.e., for \textit{query}_Q1, \textit{query}_Q2, \textit{query}_Q3, unless they are picked by the same node at different times, i.e., recycler [84, 83] reduces them to a single execution.

Exploiting result caching, the \textit{opt\_split()} can be improved to extract these type of instructions into a separate sub-function. The instructions to be extracted are the ones which read dimension data and prepare it through selections and aggregations, before joining it with the fact data. By applying this rule, a new sub-function \textit{query}_Q0 is created and \textit{query}_Q2 (cf., Figure 6.8a) is replaced by \textit{query}_Q2' (cf., Figure 6.9).\textsuperscript{13}

The resume calls in \textit{query}_Q1', \textit{query}_Q2', and \textit{query}_Q3' make explicit the re-use of \textit{query}_Q0 result. The mid-result is now available to also be used by other queries. The dimension data preparatory step can be re-used across several queries to

\textsuperscript{13}The same happens for \textit{query}_Q1 and \textit{query}_Q3.
begin query Q2();
X3 := request("t","id",2);
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);
X25 := pin(X3);
X28 := join(X25, X9);
unpin(X3);
X31 := reverse(X28);
X34 := leftjoin(X31, X25);
dacyExport("query.Q2",X34);
end query;

begin query Q3();
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);
X26 := pin(X4);
X29 := join(X26, X9);
unpin(X4);
X32 := reverse(X29);
X35 := leftjoin(X32, X26);
dacyExport("query.Q3",X35);
end query;

(a) Sub-computation query.Q2. (b) Sub-computation query.Q3.

Figure 6.8: MAL plan after opt_split optimizer.

improve throughput. Workloads like TPC-H style are the ones that can benefit most from this re-utilization.

Another scenario is the SkyServer workload [137, 67]. It contains several queries to first tune search parameters, e.g., brightness and wavelength, and then use them on the search of new objects in space or track their trajectory. By extracting the tuning parameters definition into an independent sub-function, dozens of queries can re-used them and thus improve their response time.

Like the recycler architecture [84, 83], result-caching in DaCyDB is especially suitable for applications with prevailing read-only workload and relatively expensive processing, such as data analytics and decision support. However, the result-caching at the sub-function level instead of the operator level makes DaCyDB more robust to low data volatility. The trade offs and optimization of this solution are however, part of future research.
begin query.Q0();
\begin{align*}
X5 & := \text{request}("c","tId",1); \\
X11 & := \text{request}("c","tId",2); \\
X12 & := \text{request}("c","tId",3); \\
X17 & := \text{pin}(X5); \\
X18 & := \text{reverse}(X17); \\
\text{unpin}(X5); \\
X19 & := \text{pin}(X11); \\
X20 & := \text{reverse}(X19); \\
\text{unpin}(X11); \\
X21 & := \text{pin}(X12); \\
X22 & := \text{reverse}(X21); \\
\text{unpin}(X12); \\
X9 & := \text{pack}(X18,X20,X22); \\
dacyExport("query.Q0",X9); \\
\end{align*}
end query;

begin query.Q2();
\begin{align*}
X4 & := \text{request}("t","tId",3); \\
X26 & := \text{pin}(X4); \\
X9 & := \text{resume}(Q0); \\
X29 & := \text{join}(X26,X9); \\
\text{unpin}(X4); \\
X32 & := \text{reverse}(X29); \\
X35 & := \text{leftJoin}(X32,X26); \\
dacyExport("query.Q2",X35); \\
\end{align*}
end query;

(a) Sub-computation query.Q0.

(b) Sub-computation query.Q2'.

Figure 6.9: MAL plan to enhance result caching.

6.4 Complex queries

The trends for real-time data mining and business intelligence applications show that queries are becoming more complex and executed on extremely large volumes of data. For these type of applications the division of a query plan into sub-plans for remote execution restricts the application to scale by creating hot-spots with certain operators such as joins. The problem is amplified for data sets which do not fit into the star and snowflake schema, this is, are not organized into fact and dimension columns.

A different query parallelism of finer granularity is required to reduce hot-spots and at the same time exploit large multi-core processors such as GPUs. The dynamic slicing approach presented in Section 6.2.2 is here improved in an attempt to have optimal balanced resource utilization. Instead of spreading the query plan at the sub-function level, queries are spread at the operator level. For example, the sub-function plan (query.Q1') in Figure 6.3, the join in each block is spread among three nodes. The result is packed at the fourth node. The idea has been explored through a prototype for a distributed join algorithm on a ring, cyclo-join [55, 54].

6.4.1 Cyclo-join

The cyclo-join explores the effortless and efficient data movement to design a distributed join operator. 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 (Si) locally using a commodity in-memory join algorithm. For both joins, the sorted and the hash building is done in a distributed fashion and re-used by other nodes.

The effect of skewed input data on the cyclo-join mechanism was studied by generating input tables according to a Zipf distribution with varying Zipf factors z. For various z values input data of size |R| = |S| = 412 MB (36 million 12-byte tuples) was generated. For each generated instance the join R ⋊ S was run once on a single host and once on a cyclo-join ring that consists of six hosts. Figure 6.10 reports the execution times that we measured for the join phase of the partitioned hash join. The setup phase is omitted in this graph since it is unaffected by the data skew.

For Zipf factors of z = 0.6 and greater, the exponential increase of the number of duplicates in the data sets begins to have a noticeable effect on the execution time of the in-memory hash join. This is not a surprise: the increasing number of hash collisions lets hash join slowly degrade toward a nested loops-style evaluation.

The distributed join (white bars) can handle the increasing skew appreciably better. While the processing of uniformly distributed data cannot benefit from a cyclo-join-based execution, Figure 6.10 shows a five-fold advantage of cyclo-join for input data with a skew of z = 0.9.

The benefit comes from two sources. First, the ring buffer mechanism balances
differences in the execution speeds of the participating hosts. Thus, a host that is stuck in a chunk of data with a high number of duplicates does not immediately slow down all other nodes. A predecessor only has to start waiting once it has fully consumed all data in its ring buffer.

Secondly, distribution leads to a better use of CPU caches. Cyclo-join chops all input data (in particular the inner join relation \( S \)) into pieces. Thus, even in the presence of skew, individual partitions within the hash join are less likely to exceed the size of the CPU caches and the join phase can perform more work from within caches.

The concept behind this prototype can be applied to other operators turning a ring of nodes into a powerful DDBMS for complex queries. On one, or more DaCy-cycles, exploiting nodes cache for data locality, a set of independent instructions are executed in parallel. At the first cycle raw data is loaded for processing, such as filtering, and the output directly used as input for another set of operators, such as aggregations, or joined with another raw data. Cycle after cycle, the data passes the different stages that compose a complex query.

With the DaCy-cycle as time unit, the execution process can be stretched into a time line with operators overlapping or interconnected in chain representing sequence of filters, aggregations, joins, etc., and the final result being returned in the last DaCy-cycle.

### 6.4.2 Distributed state machine

The number of cycles is dependent on the number of partitions and the number of nodes which both are dynamically adjusted to achieve the best scalability. Furthermore, an operator can also be executed using several cycles to exploit recursion, such as tail recursion. Tail recursion is a special case of recursion semantically equivalent to the iteration constructors used to represent repetition in programs. Hence, tail-recursive programs can be compiled as efficiently as iterative programs. It turns them more efficient and in addition their definition more clearer. The simplicity of using tail recursion becomes more compelling when the number of instructions and the complexity of their call graph increases.

Hence, a complex query can then be simply compiled into a large finite-state machine with separate tail-recursive functions instead of using a single huge query plan. The parallel execution plan can thus be represented using sophisticated graphs to represent all possible parallelism strategies. For example, DLP-graph (data, precedence and loop dependencies graph).

Contrary to the simple representation of intra-operator parallelism, this is, data dependencies graph (D-graph), a DLP graph models all possible parallel execution strategies by containing the precedence dependencies, such as an operator must be
terminated before another operator can start, though no data dependency is involved. Furthermore, they also allow explicitly representation into the parallel execution plan of loop dependency. A loop dependency indicates that a sequence of operators must be repeated as many times as there are available partitions.

With the parallel execution plan defined as a DLP graph, DaCyDB would not be anymore bounded as other systems to static parallelism strategies. It could adopt solutions that integrate run-time control mechanisms. Based on the execution costs, the operators execution could be re-ordered to correct load imbalance, optimizer estimation errors, etc.

As in XPRS [79, 135], a special choose operator could be introduced to at runtime to choose different execution alternatives. Another option would be the exchange operator of Volcano. It includes some kind of control, as it can dynamically re-estimate the degree of the intra-operator parallelism [29].

The search space is enormous and old ideas could now be revived for the new hardware trends. The study of the different optimizations as well as the integration of a functional front-end language to define the DLP-graph such as FunSQL [30] is part of the on-going research not covered by this dissertation.

6.5 Dynamic resources allocation

The workload is not stable over time. The search space might have shrunk or the computations became more complex. The immediate consequence is that the initial Data Cyclotron ring size requested by the user may not be optimal in terms of resource utilization and performance. For example, having more nodes than strictly necessary increases the latency in data access while CPU cycles are not exploited. Contrary, having too few nodes reduces throughput. The challenge is to detect such deviations and dynamically adapt the ring structure.

The first step was taken with dynamic plans (cf., Section 6.2.2). They are used to on the fly adjust the query execution plan to achieve efficient intra-node resource utilization. For efficient inter-node resource utilization, DaCyDB uses a pulsating ring, one that adaptively grows and shrinks to find the lowest number of nodes comprising a ring to seek the optimal point for throughput.

A pulsating ring in a self-organizing way, based on the amount of queries, data flowing and nodes load, they shrink or grow. It grows to achieve higher throughput,

---

14 Extension to SQL which allows application developers to decompose logic into functions (with multiple input and output parameters) and assign functions to bind intermediate results of SQL queries to variables using a static single form (as in functional languages).
but the growth is contended by the data access latency. It does not rely on a central co-
ordinator, all decisions are independently done at each node. It makes the architecture
flexible and robust to scale and together with intra-node dynamic resource allocation
turns DaCyDB more cost effective than MapReduce- inspired solutions.

Contrary to MapReduce paradigm, the right amount of resources and their efficient
utilization is all done by DaCyDB and not through static configuration parameters. The
user is thus released from the complexity to specify the accurate amount of resources
for complex computations. If not accurate it leads to inefficient resource utilization
(cf., Section 2.3.2).

"For complex computations MapReduce solutions looses resource utilization ef-
ficiency. The inefficiency is due to three drawbacks, single node resource usage un-
balance, reduce slot hoarding, and resource allocation unbalance within a job [142].
... Last, resources are allocated by a static configuration which does not consider the
system dynamical load and job requirements. The number of slots is always fixed inde-
dependently of the number of mappers or reducers. Furthermore, the cluster might have
too many map slots, but not enough reduce slots, and vice-versa. "

6.5.1 Pulsating ring

In DaCyDB computations are not spread based on data assignment, but purely based
on node characteristics allowing the system to maximize its resource utilization. They
move anti-clockwise waiting to be picked up by a node with ample resources. The
priority of a computation increases with each transverse of the ring and not be picked
for execution. A priority based on the number of cycles works as a timeout for their
stay in the ring. Each node selects the queries with highest priority. In case of lack of
resources, the node postpones the selection until further release of resources.

When the nodes cannot pick more computations up from the ring it means the nodes
are I/O bounded or CPU bounded. In case they are CPU bounded, the ring needs to
grow. In case it is I/O bounded, the decision depends on the storage ring load. With
low storage ring load, it means the ring is too large, this is, the data access latency is
taxed by the large number of hops the data chunk does to reach the destination. With
storage ring overloaded, tension of the node’s buffers delays the load of data chunks,
and thus data access latency is increased.

Any node can request the insertion of an extra node or leave the ring if its resources
are not being exploited over a satisfying threshold. In case of an extension, the ring is
increased step wise until the data access latency increases the idle time of the CPUs. In
case of a contraction, the nodes stop leaving the ring once they start using the neutral
zone of their DaCy storage for data in transit. With homogeneous hot-set, it means
the storage ring is on its optimal utilization and without nodes with tension on their buffers, i.e., there is not groovy nodes.

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 as represented in Figure 6.11 A). A leaving node simply notifies its predecessor node and its successor node to make a direct link and to ignore the leaving node from there on. This elastic deformation is bounded by the initial ring size and the number of nodes available for insertion.

More flexibility is obtained if we use the delegate data chunk ownership model presented in Section 5.4.3. On a ring extension a node with high rate of data chunks 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 persistent storage for the data set.

6.5.2 Pulsating ring algorithm

In a pulsating ring the nodes can be added or removed in two ways, one node at the time or several nodes within a single step. In both cases, a lock is used to avoid concurrent structure deformations on the same node. It is assure that no data is lost on the process and it is transparent to all other nodes, i.e., the remain nodes do not notice that a node joined or left the ring.
Lock acquisition.

To leave the ring or to request a new node, a node must obtain the pulsation’s lock. The pulsation’s lock is a message that flows around the ring and it contains a status, locked or unlocked, and a list with the node’s addresses waiting to be added. The node that changes the status from unlocked to locked is the one that has the right to modify the ring structure. Until the status is changed back unlocked no other node can modify the ring structure, i.e., until the lock is released.

For a contraction, it is the left neighbor of the leaving node who releases the lock. It releases the lock after receiving the last chunk from the leaving node. For an extension, it is the new node who releases the lock.

Ring contraction.

A leaving node C, after getting the lock, puts is address in the end of the list, tells its right neighbor A to connect and start forwarding data to B, i.e., C’s left neighbor. In mean time C forwards the remaining data chunks on its buffers to B, i.e., B receives data from A and C as shown on Figure 6.11 B). Once C is done with the forwarding, it informs B to shutdown the connection and release the lock. From that point on B only receives data from A.

A node is allowed to leave if its neighbors are not leaving at the same time the ring. Hence, in the extreme case, a ring can shrink one third of its size within a single step. For small rings, to avoid buffer contention due to the double data forwarding (Figure 6.11 B) ) only one node at the time is allowed to leave the ring.

Ring extension.

Once a node A gets the lock, it informs the first node on the list, node C, to connect to A and A’s left neighbor, node B as pictured in Figure 6.11 A). C initializes its internal structures and routines based on information shared by A. Once ready, it informs A and B to update the TCP sockets or the RDMA context of their transmission/reception threads.

Due to the fact a new node is inserted between the requester and the requester’s left neighbor, and the nodes list flows around, the ring extension can be improved to allow multiple extensions one a single step. The pulsation’s lock type is modified to become a shared-exclusive lock, i.e., only one node at the time can leave the ring, but more than one node can request an extension. Hence, race conditions with leaving nodes are avoided and the ring, in the extreme case, can duplicate its size with a single step as pictured in Figure 6.11 C).
6.5.3 Large ring optimization

Workload with large hot-sets tend to request a large ring. The extension to a large ring is normally bounded by the data access latency. The latency to access the less popular chunks and fresh chunks may become a performance hindrance.

As already observed in the Broadcast disks approach [4], 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 Data Cyclotron architecture to achieve the same goal, although, we take another course.

With the warm-state introduce in Section 4.4, we have partially introduced the concept of seep lines, this is, the data chunk forward is based on a priority instead of **FIFO** order.

"The Data Cyclotron at load time assumes that all data chunks to be loaded have the same probability to become standard or even in vogue data chunks. However, not all data chunks are classified as such. In case the loaded chunk is an unpopular one low data access latency is assured for few computations, but overall it downgrades throughput. To overcome this issue we propose pre-warming up phase before the data chunk load into the hot-set. In this phase, the data is in the warm state, an intermediate state between cold and hot."

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. Since the data will only be available at a subset of the nodes, it could lead to throughput degradation, especially for skewed workloads.

To circumvent the problem, we introduce an inner-ring represented as a square in Figure 6.12. The *in vogue* data 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.

A inner-ring is simplified version of a chordal ring, a simple ring with cross or chordal links between nodes on opposite sides. With different dimensions a chordal ring is used to define several virtual rings to speed up data forwarding of a single hot-set in a large ring. In case the hot-set can be decomposed into disjoint, or partially disjoint, hot-sets a large heterogeneous ring is not anymore the right for efficient re-

139
source utilization and low response time. The ring should be fragmented into smaller overlapping, or independent, pulsating rings. The agglomeration of all these rings is called the Data Cyclotron Mesh (cf., Section 6.6).

### 6.6 The Data Cyclotron mesh

The Data Cyclotron has an innovative architecture with a large scope of scientific challenges. In this section we discuss an even more flexible architecture, the Data Cyclotron Mesh. It is an aggregation of pulsating rings Figure 6.13. They are used in the Data Cyclotron to provide an outlook for query processing workloads that can not be addressed with database sharding and map-reduce. Workloads composed of thousand of queries grouped by common interest on a database sub-set or specific resource requirements.

With the computations crumbled into sub-functions (cf., Section 6.2), DaCyDB exploits the Data Cyclotron mesh to allocate each sub-functions group to a different ring. With a priority policy to pick computations for execution not only based on the age, but also based on the request list and resources requirements, such as virtual memory size or different processing unit, DaCyDB allocates different rings to improve cooperative work, result caching, low data access latency, and in some cases, exploit rings with specialized hardware.

#### 6.6.1 Definition of new rings

The definition of a DaCy mesh starts with an initial ring $R_1$. In the initial ring, $R_1$, each node observes the request list of each computation passing by in search for access
patterns over the data under its ownership. From the access patterns, a node $N$ derives if its data is only used with a small sub-set of the hot-set, e.g., $D_2$. If yes, it extracts $D_2$ from the hot-set and moves it to another ring, $R_2$. During the process no other node is allowed to build a new ring.

Before the data delegation, $N$ requests a new node to initialize $R_2$ and, without leaving $R_1$, creates a two nodes ring. $N$ becomes the bridge for the data and computations between the two rings. More than one bridge can exist, if the disjoint set was detected by more than one node. The ring $R_2$ is then defined by establishing connections between those nodes.

From the access patterns and the computations age, a node is also able to identify a complete new hot-set, $D_3$. If a set of computations is not collected for execution due to a disjoint list of request compared to the actual hot-set, a node ($N$) instead of requesting a ring extension, it defines a new independent ring, $R_3$. It then moves the computations to the new ring. The computations requests will trigger the move of data from $R_1$’s cold-set to $R_3$’s hot-set. In case the data is not present yet in $R_1$ cold-set, iterative loading is used to load the data from source nodes.

Another example, is the type of resources, such as graphical processing unit (GPU), required by a group of instructions. It is known the GPUs are more suitable for computations involving matrix and vector operations. Using decision models [28] to identify instructions which will benefit from the GPU characteristics, DaCyDB can pack them into independent sub-functions. If a high number of this type of sub-functions, instead of a ring extension, the a new ring, $R_4$, only composed of nodes equipped with GPUs, can be allocated for more efficient resource utilization.
6.6.2 Inter-ring interactions

With several rings co-existing in the same cluster, computations and data need to be re-directed to the correct ring. To keep the architecture decentralized we add an outer-ring. The outer-ring is composed of all nodes which were the first node of a ring. These nodes are designated as primary-bridge nodes.

Hence, no routing tables or DHTs are required to re-direct traffic between two ring indirectly connected. Once a request cannot be fulfilled in a specific ring, the request is posted in the outer-ring to be picked up by a primary-bridge node which has the requested data in its catalog. In case the request returns, the data is requested from the source nodes. The outer-ring is also used to propagate data from one ring to another through ownership delegation or request.

At any point in time, the mesh consists of numerous overlapped rings. Each ring is focused on a given subset of the workload. Rings appear, grow/shrink, and disappear as the workload changes. Effectively, system nodes, except the primary-bridge nodes, adaptively hop from one ring to another to accommodate the workload needs. A ring grows when more resources are needed and shrinks when more than necessary resources are available. Once a ring shrinks completely, i.e., it is only composed of a primary-bridge node, it disappears.

The Data Cyclotron Mesh is analog to a crowd in sociology. The continuous adaptation of a ring to the workload demands, without central coordination, behaves the same way as group of persons, i.e., they join an leave the group base on its popularity. If each ring can be seen as a group, the Data Cyclotron Mesh is then seen as crowd organized by groups.

Each group, i.e., each DaCy ring, focus on a specific workload. Over the time, rings become more popular than others, i.e, they grow due to a workload increase. As groups, rings disappear over the time, others split into sub-rings or even merge to form a bigger ring. The interest of the workloads defines the shape and organization of the crowd, i.e., the Data Cyclotron Mesh. The model exploits the independence and autonomy of each individual where everyone, based on the workload flow, works for the same goal, high throughput. As in society, not all individuals are highly satisfied, but a global satisfaction is achieved.

6.7 Fault-Tolerance

In the ring topology the failure of a node interrupts the data flow among the nodes and if a replica does not exist, the data cannot be recovered. To cope with this issue the Data Cyclotron provides two fault-tolerance models: in-ring and out-ring mode. The
fault-tolerance model to be used depends on the type of workload. For workloads requesting data from a data source which is always online and has a stable connection with one or more nodes in the ring, the out-ring mode is used. Otherwise, the in-ring mode is used.

**Data replication.**

In both models a data chunk has three replicas. The location of the replicas is what distinguishes a model from the other. For out-ring mode, replicas are located at the storage nodes and all of them are used for efficient parallel data access. At the storage nodes data replication is done with multi-primary scheme (called multi-master in the database field). On these nodes, the data is in read only mode. Hence, there is no need for a distributed concurrency control such as distributed lock manager.

For in-ring mode, replicas are spread around the ring, i.e., among the node’s successors. The data distribution attempts to have two partitions within the same rack and one out-side of the rack. It exploits the fact that nodes from different racks are mixed (cf., Section 5.4.3). It makes DaCyDB robust against a full rack crash. Furthermore, only one replica is set as primary replica. The node with the primary replica is the one responsible to load its content into the storage ring to fulfill requests.

**Connections re-establishment.**

For both models the connections are re-established using the Chord lookup protocol [132]. In Chord lookup protocol the nodes keys are arranged in a circle that has at most $2^m$ nodes [145]. The circle has IDs/keys ranging from 0 to $2^m - 1$. Each node has a successor and a predecessor. The successor to a node (or key) is the next node (key) in the identifier circle in a clockwise direction. The predecessor is counter-clockwise.

Since the successor (or predecessor) node may disappear from the network (because of failure or departure), each node records a whole segment of the circle adjacent to it, i.e. the $r$ nodes preceding it and the $r$ nodes following it. This list results in a high probability that a node is able to correctly locate its successor or predecessor, even if the network in question suffers from a high failure rate.

**Data recovery.**

With the connection established, the next step is to reload all data lost with the failed nodes and restart their computations. For both modes, when a node fails the requests report back data missing. If the data is not in the hot-set, warm-set, neither cold-set, it will come from storage nodes or from a node in the adjacent list.
For out-ring model, using iterative loading, only data in fault is re-loaded from the storage nodes. While loading, the data is accessed by the existent nodes and a new node can be added on-the-fly to become the new owner of the flowing data. With iterative loading, the recovering is an incremental process and it does not affect data traffic on other nodes. Furthermore, there is not replica recovery, i.e., creation of a new copy. The replicas management is done at the storage nodes through traditional replication techniques [74].

For in-ring mode data is retrieved from the first node in the adjacent list, i.e, one with a secondary replica. It becomes the primary replica and starts to fulfill requests. In mean time a new node, or a successor, is added to the adjacent list. It gets a secondary replica.

The management of secondary replicas is done through heartbeats. The secondary replicas sent heartbeats to the ring to inform the primary replica they are alive. In case one stops sending heartbeats, the primary replica is responsible to re-establish the missing replica.

Computations recovery.

When a computation is picked for execution it does not stop its journey. It hops more three nodes to be replicated. They are call clones. In case of failure, the node in the adjacent list used to recover the data, restarts all clones it has.

In case the computation has not settled down, i.e., it is not in execution, and the node that holds it on its buffers crashes, the computation cannot be recovered. The same is not applied for sub-computations. They are recovered through time-outs in the resume() calls. A resume() call contains the sub-computation plan for matching, therefore, when a time-out occurs and the mid-result was not retrieved from the storage ring, the sub-computation plan is sent back to the ring for re-execution (cf., Section 6.2.1).

" The dacy_export is used to publish the function plan (FP) and the result into the ring as an ordinary data chunk, called mid-result 15. For query Q1 the result is column X33, see instruction B) in Figure 6.2b. ... The merge operator is then moved into another sub-function (query Q) proceeded by one or more resume() calls depending on the number of dependencies on other sub-functions, see Figure 6.2c. A resume() call collects a mid-result from the storage ring. It uses FP to identify the mid-result. "

15The name mid-result is used to distinguish this type of intermediate results from the ones produced within a sub-function execution
6.8 Green architectures

In the past decades, the database and distributed data processing communities have designed and developed a wealth of methods to optimize response time and/or throughput. Energy consumption was however never considered. Economical and environmental factors are requiring to see energy as a critical performance metric for data processing. Energy awareness is opening a wide range of research challenges [100]. New DBMS architectures, such as EcoDB [48], started to consider using energy consumption as a first-class metric in a DBMS when planning and processing queries. On the horizon is the definition of energy efficiency metrics for established benchmarks such as those defined by TPC [66].

Nonetheless, the efforts to improve query response time are still very valuable. In most of the situations, faster query execution results in lower energy consumption. However, a more energy efficient data center cannot be realized if there is not energy efficient applications to run on top of energy efficient hardware and networks [36].

From the hardware research community, energy management is already an important metric and design criteria for modern data center management and planning [17]. They are trying to circumvent the problem that the energy consumption of current hardware is not proportional to the actual load on the machine [17, 50].

With DaCyDB we are trying to contribute to this green movement. The key feature of DaCyDB, continuous data movement, has raised some concerns about its energy efficiency. With a thoughtful analyze of ongoing research for energy efficiency, we will show how the DaCyDB architecture has the ingredients to be qualified as a green architecture. Its extensibility for new ideas also grants the conditions to introduce new techniques for energy efficiency.

Efficient resource allocation.

With iterative data distribution (cf., Section 6.2.4), dynamic plans (cf., Section 6.2.2) and pulsating rings (cf., Section 6.5.1), DaCyDB attempts to process as many computations as possible in a single node and have short rings. For large hot-sets, the ring is kept large enough to keep the all hot-set flowing, and thus avoid constant data reloads. A reload often reads data from disks which are known for their energy inefficiency [100].

In case nodes start to be under utilized, they leave the ring to save resources. In case the data access latency increases with the removal of nodes, the ring does not shrink. In this situation, the technique PVC (Processor Voltage/Processing Control) [100] is used to reduce energy consumption. PVC exploits the new state-of-the-art processors.

\[16\text{www.tpc.org}\]
They take commands from software to adjust their voltage/frequencies to operate at different levels than their peak performance. By reducing the frequency of the nodes processors, under-utilized nodes can remain without much energy consumption since all data forwarding is conducted by RNIC using RDMA (cf., Section 5.4) which is more energy efficient than TCP Sockets [104].

The power savings of RDMA are achieved by minimizing the interactions between the network adapters, CPUs and memory. Compared to TCP/IP, RDMA requires much fewer CPU cycles for protocol processing and also generates less memory bus traffic (cf., Section 2.6.2), both of them contribute to its power savings.

“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) [13]. ... 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.”

DaCyDB also uses PVC for rings with slow networks, i.e., computations are network I/O bounded. Instead of having processing peaks at the nodes, i.e., for each data chunk arrival the processor is taken to its maximum frequency, they process data with lower frequency to align with the data arrival rate without increasing response time.

**Topology.**

The topology simplicity is another feature of DaCyDB for energy efficiency. With a mesh of pulsating rings the network traffic at the switch is relative simple compared to a 2D mesh or partial connected mesh. In DaCyDB the data is transferred in big chunks, constant routing pattern and without bursts. Hence, the DaCyDB traffic requires low synchronization and small amount of resources, such as buffers, to solve contentions (cf., Section 3.2).

“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.”

The data center to host DaCyDB can trade complex switches for simpler switches,
and thus the number of watts spend per each GB transfer be reduced. The idea has been explored for on-chip networks. Authors in [26] use a topology similar to Data Cyclotron Mesh (cf., Section 6.6). They use an hierarchical ring topology which consists of small local rings connected via one or more global rings. The design obtains the advantages of both mesh- and ring-base designs, and avoid their shortcomings.

Ring networks use very simple routers, which reduces energy and die area, and eases design and validation, in some situations, performance suffers with scaling (cf., Section 5.4.5). Mesh networks scale relatively well, however, they require large, complex, energy-inefficient routers. Authors [95], for on-chip networks, showed rings achieve very good energy efficiency at low-to-medium core counts, or when network load is low enough that the ring does not become a bottleneck.

The work on on-chip networks emphasizes the fact for scalability energy efficiency, hierarchical ring composed of small rings should be used. When composed of large rings, scalability becomes an issue. DaCyDB uses inner-ring to circumvent the problem (cf., Section 6.5.3).

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

Furthermore, with the inner-ring data compression can be used to reduce network traffic, and thus increase energy savings. All data loaded into the inner-ring is compressed. When forwarded to the outer-ring, only the popular data chunks are decompressed to avoid the cost of decompressing data in most of the nodes. The unpopular data is only decompressed at the nodes where it is used, releasing network bandwidth to transfer more data.

Typical compression algorithms, such as gzip, are not the ones to be adopted since they increase the net energy when compression is applied before transmission [16]. Energy-aware lossless data compression [16] used for wireless networks is one of the paths which can be explored since it is an energy-aware solution and efficient as typical compression algorithms. The authors [16] used one compression algorithm on the transmit side and a different algorithm for the receive path. They have shown on their prototype that by choosing the lowest-energy compressor and decompressor, overall energy to send and receive data can be reduced by 11% compared with a well-chosen symmetric pair, or up to 57% over the default symmetric zlib scheme. The choice of right compressor and decompressor for DaCyDB context is part of ongoing research and thus, not cover by the content of this dissertation.
Recent studies have focused on redesigning data center server clusters with low cost, low-power wimpy nodes [10, 86] because they are relatively well-balanced [127]. These type of nodes are more energy efficient due to their lowend CPUs and low-power components. On the other hand, they lag far behind traditional nodes in performance. Hence, to replace a small cluster of traditional nodes, such as Xeon\(^{17}\) nodes, a larger cluster of low-end nodes is required [101]. This type of clusters are the ones used by MapReduce frameworks which are mainly designed to scale out (cf., Section 2.3.3). However, complex computations exhibit disproportionate scale up characteristics which potentially makes scale-out with low-end nodes an expensive and lower performance solution. Authors in [101] have shown that for data-intensive workloads, a TPC-H workload, large wimpy node clusters suffer from poor scale up effects, and thus are potentially slower and a costlier solution compared to Xeon clusters.

Wimpy clusters are more affected by a diminishing return scale up effect than a smaller traditional cluster [101]. The work shows that in real (as opposed to ideal) scale up environments, price/performance degrades as the scale up factor is increased. It gets more expensive to achieve the same level of performance. Nevertheless, the experiments also showed that using only Xeon nodes was not the direction to be taken. A hybrid solution is the one allows to have low response times and energy consumption [101, 37]. Hybrid cluster deployment strategies, job scheduling, and scale up analysis were mention as interesting avenues of future research [37].

DaCyDB aligns with this line of thinking. It is tailored for complex computations through its seamless transition between the scale up model and the scale out model (cf., Section 6.2).

"The DaCyDB provides a seamless transition between a scale up solution to a scale out solution by having both in harmony. Based on the resources of each node the computation on-the-fly has the flexibility to explore both models in different degrees. The scale-up model is used for balanced resource utilization and explore data locality using the nodes cache. The scale-out model is used to spread the execution of complex queries across the cluster. With heterogeneous hardware, both models complement each other to achieve high throughput and optimal resource utilization."

Furthermore, with the ring as logical topology, wimpy nodes and nodes like Xeon nodes can co-exist in the same ring. The computations are picked by the nodes based on their resource utilization (cf., Section 5.4.4). In case a computation is too slow, it

is decomposed and executed by several nodes through the *dynamic plans* optimization (cf., Section 6.2.2). If not possible the decomposition, the computation is restarted somewhere in a bigger node.

Instead of restarting the computation, the computations status can be frozen and sent for execution in another node, for example, a node with ample CPU cores, a GPU or higher amount of remote memory for virtual memory. Only the intermediates and the remain plan execution are transferred; the data needed will pass by upon request. Hence, in DaCyDB, using compact rings with heterogeneous nodes, complex computations are broken up in pieces for efficient processing and energy awareness.

**Query scheduling.**

With queries flowing in the storage ring, DaCyDB has the capacity to explore Query Energy efficiency by introducing explicit Delays (QED) [100]. With QED, queries are explicitly delayed for workloads in which there are often common components across different queries (and delays can be tolerated). Hence, the queries build up in the queue to be executed as batch. The authors in [100] had energy saves up to 54% using a TPC-H workload on MySQL cluster and commercial DBMS. However, the performance decreased by 43%. The major reason was the inefficiency to exploit the commonalities between the queries. Furthermore, they were merged into a single query and then the result has to be split which is an expensive procedure.

DaCyDB tackles the problem from a different angle. Queries are split into sub-functions and DaCyDB uses them to find commonalities among the queries, i.e., through *result caching* (cf., Section 6.3) DaCyDB reuses sub-functions result avoiding the execution of hundreds of instructions. Furthermore, with sub-functions dependencies defined through a dependency graph, each query result can be built independently without extra cost.

Another energy saving is on disk accesses to retrieve data for the queries. With priority policy to pick queries up for execution, defined for the Data Cyclotron mesh (cf., Section 6.6), the hot-set definition is aligned with the queries batch. Hence, with less shifts on the hot-set, less disk reads have to be performed to load new data into the ring. Disks reads, specially random reads, are known to be energy inefficient [100].

"With the computations crumbled into sub-functions (cf., Section 6.2), DaCyDB exploits the Data Cyclotron mesh to allocate each sub-functions group to a different ring. With a priority policy to pick computations for execution not only based on the age, but also based on the request list and resources requirements, such as virtual memory size or different processing unit, DaCyDB allocates different rings to improve cooperative work, result caching, low data access latency, and in some cases, exploit
6.9 Where does DaCyDB stands compared to other solutions?

The vision for DaCyDB presented an novel and dynamic architecture to cope with the challenges of current complex workloads for large data-sets while it offers efficient resource utilization on modern hardware. It is not a one-size-fits-all architecture. It is designed to efficient at a particular area in the data processing line. From data collection until the result presentation, several data processing stages are crossed. Hence, for each of them we identify which architectures are more suitable than DaCyDB or when DaCyDB stands as the right system to be used.

In general, the data processing is divided into three stages: data collection, data preparation, and data presentation [58]. The last two are the ones for which the world of data management has payed more attention. Several solutions, such as high performance data warehouses, Hadoop implementations, extreme performance, or data integration technologies, have emerged to cope with the challenges of these two stages. Each solution has its strengths and weaknesses depending on the data processing stage.

In general, the MapReduce frameworks were designed to be efficient on the data preparation stage while DaCyDB was designed for the data presentation stage. Nevertheless, both solutions can also be adapted to somehow be efficient on others data processing stages. For example, Yahoo considers Pig [60] more suitable for the data preparation phase while Hive [139] is more suitable for the data presentation phase [58].

6.9.1 Data preparation stage

The data preparation stage is normally classified into three categories, pipelines, iterative processing, and research [58]. In the first category the pipeline brings in the data feed, such as logs from web-servers, cleans it and transforms it. A MapReduce framework is without doubts the most suitable for it.

In the second category there is usually one very large data set that is maintained through incremental updates. For example, a huge graph where each node is a page and the links the relation between nodes based on the page content. An update resumes to the integration of recent news.

The DaCyDB could fit into the model by having the update data flowing in the ring to update data stored in the cold-set. However, for this scenario latency is more important than throughput which makes the MapReduce paradigm more efficient [45].
In the third category petabytes of data are brought in and researchers quickly write a script to test a theory or gain deeper insight. The data is not always in a nice, standardized state. It is a typical preparation stage on an eScience scenario.

In this scenario the ELT model is the preferred one to load the data, this is, the data is extracted and loaded directly into the data warehouse and the transformation occurs already in the data warehouse. For example, the raw data sets with un-structure data, e.g., EBAY logs, are data-sets that cannot be a priory structured for further analyzes. All relevant information is extracted, combined and analyzed at runtime.

The process requires a full scan of the entire database which is suitable for MapReduce. However, the DaCyDB architecture stands here as promising architecture to compete with MapReduce frameworks. The elements of a ring would scan the unstructured data and turn it into semi-structure data for analyses. Breaking up the web-click records in essentially key-value sets which are stored in binary columns floating around the DaCyDB ring.

The ring can be composed of nodes with different functionalities. Given the inherent cost of parsing, several may be chartered to read and tokenize the data, while others could be chartered to hunt for specific patterns or perform a learning algorithm. The data needed remains available in the ring until all parties interested have seen it, where after it can be garbage collected or stored for later use.

With iterative load and the possibility to apply different levels of integrity (cf., Section 6.2.4), sub-sets are loaded into the ring for inspection and be correlated with other data sources. The architecture allows an incremental database exploration, where the amount of processing can be controlled by the priority and size of the individual rings.

### 6.9.2 Data presentation stage

The data presentation stage is classified into two dominant use cases, business-intelligence analysis and ad-hoc queries [58]. In the first case, users connect the data to business intelligence (BI) tools to generate reports or do further analyses. In the second case, users run ad-hoc queries issued by data analysts, decision makers, or scientists.

In both cases the ETL model is often the one used to load the data, this is, the data is extracted to a staging database, transformed in the staging database, and loaded to the data warehouse. The MapReduce is often used to transform and load the data. Furthermore, its explicit and scalable dataflow paradigm has become popular for this applications in favored the traditional high-level declarative approach SQL due to its simplicity.

However, the extreme simplicity leads too much low level hacking to deal with the many-step, branching dataflows that arise for complex computations and standard op-
erations such as joins which are repeatedly code by hand [60]. Pig and Hive are the best examples to move the user away from this low level hacking and complex computations optimization. However, they are limited by the initial design of MapReduce for data distribution (cf., Section 3.6), complex computations (cf., Section 2.3.4, and resources utilization (cf., Section 2.3.2).

Furthermore, the relational model and SQL are the best fit. "Indeed, data warehousing has been one of the core use cases for SQL through much of its history. It has the right constructs to support the types of queries and tools that analysts want to use. And it is already in use by both the tools and users in the field" [58]. DaCyDB through the relation operators from MonetDB and the flexible architecture of Data Cyclotron stands here as a robust and efficient solution for the challenges posed by the data presentation stage in the eScience world.

6.10 Summary

The DaCyDB realizes the idea of data movement between network nodes as an ally for improved system performance, flexibility, and query throughput. The absence of static data allocation is exploited to explore different algorithms for distributed processing. Computations are split into sub-computations and sent to the ring. Each of them settles on a different node following the basic procedures of a normal computation. They are processed concurrently and the individual mid-results are then combined to form the final computation result. Such freedom gives the grounds for an application to scale out without be bounded to any complex scheduling algorithm or data distribution scheme.

Such flexibility combined with the elegant operator-at-the-time paradigm used by MonetDB opens doors to explore query parallelism with finer granularity such as at the operator level. With cyclo-join it was shown a promising future for new distributed operator algorithms to exploit the new network trends and the new multi-core processors. Furthermore, combined with iterative data loading and distributed consistency check, the cumulative cost to answer the first queries is significantly reduced.

Dynamic resource allocation is achieved due to the fact 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. Hence, 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 upon request. This way, the load is not spread based on data assignment, but purely on the node’s characteristics and on the storage ring load characteristics. This innovative and simple strategy intends to avoid hot spots that result from errors in the data allocation and query plan algorithms. Furthermore, each node, based on the local resources utilization, decides to leave the ring or request a
ring extension. This autonomous and dynamic adaption to accommodate the workload requirements is what defines *pulsating rings*.

The pulsating rings used in the Data Cyclotron provide an outlook for query processing workloads that can not be addressed with database sharding and map-reduce. It is a new concept to design fully flexible distributed query processing architectures. Moreover, it is conceivable that multiple pulsating rings live in the same physical cluster. At any point in time such a mesh consists of numerous overlapped pulsating rings. The model exploits the independence and autonomy of each individual where everyone, based on the workload flow, works for the same goal, high throughput.

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