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 4

The Data Cyclotron Hot-set Management

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

A ranking system is used to determine where a data chunk should be, i.e., in disk, memory, or storage ring (also in memory, but in circulation). Its self-organization in a distributed manner, keeping an optimal resource utilization, replaces gradually the data in the ring to accommodate the current workload.

The ranking system is composed of a ranking metric called the level of interest \((LOI)\) and a threshold called the level of interest threshold \((LOIT)\). \(LOI\) describes how popular was and is a data chunk for the current workload. Lower is its value more likely is the data chunk to be evicted. \(LOIT\) is the borderline between the hot-set and cold-set. Each node keeps adjusting \(LOIT\) to have as many data chunks as possible in circulation and low data access latency.

Using a well known network simulator (NS-2), we explore different approaches and methods to reach the most efficient and robust solution to rank the data chunks popularity and to define the optimal hot-set. From the integration of an innovative cache management to ownership request we assure a homogeneous hot-set composed of data chunks with the highest probability of utilization. The study is supported by simulation results for different workload scenarios.
4.1 Outline

The remainder of this Chapter is organized as follows. Section 4.2 presents the hot-set management algorithm, and \textit{LOI} and \textit{LOIT} concepts in detail. Section 4.3 contains the study of hot-set management for different scenarios using a discrete event simulation. Furthermore, based on observations collected during simulation, we improved \textit{LOI} definition. Subsequently, we present an intermediate state between hot- and cold-set called \textit{warm-set} in Section 4.4. Section 4.5 presents an efficient cache management to avoid the relevant data forwarding be slowed down by the load and forwarding of unpopular data. Finally, the Chapter concludes with the presentation of a dynamic data ownership model to create an homogeneous hot-set in Section 4.6 and a summary in Section 4.7.

4.2 Level of interest

The data chunk is loaded once there is space in the storage ring and it flows as long its \textit{LOI} is above its owner’s \textit{LOIT}. It becomes a candidate to be part of the hot-set as soon as a request for its load is received at its owner \footnote{The data chunk’s owner is the node responsible to load/unload it to/from the storage ring.}.

Its \textit{LOI} is updated at each cycle completion, this is, every time it returns to its owner. The hot-set management determines the new \textit{LOI} based on the previous \textit{LOI} and global information collected during the data chunk’s journey such as the number of \textit{copies}, the number of \textit{hops}, and the number of \textit{cycles}. The variables \textit{copies} and \textit{hops} are updated at each node. The variable \textit{cycles} is only updated by the data chunk’s owner when it completes a cycle. Hence, as described in [63] the \textit{LOI} is calculated as follows:

\begin{equation}
\text{CAVG} = \frac{\text{copies}}{\text{hops}}
\end{equation}

\begin{equation}
\text{newLOI} = \frac{\text{LOI} + \text{CAVG} \times \text{cycles}}{\text{cycles}} = \frac{\text{LOI}}{\text{cycles}} + \text{CAVG}
\end{equation}

The previous \textit{LOI} for a data chunk represents the ring’s interest on the data chunk during the previous cycles. However, the latest cycle has more weight than the older ones. This weight is imposed by multiplication of the number of copies average \textit{CAVG} in the last cycle by the actual \textit{cycles} value:

\begin{equation}
\frac{\text{copies}}{\text{hops}} \times \text{cycles}
\end{equation}
New level of interest

input: the data_chunk_id, loi, copies, hops, cycles
output: forwards the data chunk or unloads the data chunk.

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

Figure 4.1: Hot-set Management

The division by the number of cycles was used to apply an age weight to the formula. The age weight decreases over the time to avoid monopolization of resources, this is, an old popular data chunk should give place to new popular data chunks. Hence, old data chunks carry a low level of interest unless renewed in each pass through the ring.

Data chunk removal.

Once the new LOI is determined, it is compared with LOIT. If lower, the data chunk is removed from circulation. If not, the LOI variable is set with the new LOI value and the data chunk is sent back to the ring. This hot-set management (represented by algorithm in Figure 4.1) is executed at the Data Cyclotron layer for all data chunks received from the predecessor node.

At each node, the runtime system derives the LOIT value using as reference the local DaCy storage load. Its adjustment is dynamic and inversely proportional to the DaCy storage load. If it is too loaded, LOIT increases. It is step wise increased until the pending local data chunks can start moving. If the data is being forwarded and loaded without delays it is decreased so the data is kept in rotation as long as possible.

A node keeps adjusting LOIT to have as many data chunks as possible in circulation, but at the same time not more than the available DaCy buffers for transit data to keep the data access latency low. The precision of LOI to rank each data chunk and the dynamic LOIT adjustment is analyzed and improved in the coming sections.
4.3 Simulation

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

The simulation scenario has a narrow scope. It is primarily intended to demonstrate and stress-test the behavior of the Data Cyclotron internal policies. Moreover, higher scale experiments are not recommended for this type of simulators due to some internal overflows in the packet transmission and synchronization\(^3\). Further study for a bigger scope, using the conclusions from this Chapter, is conducted with a full functional Data Cyclotron on a real-size cluster (cf., Chapter 5).

The base topology in our simulation study is a ring composed of ten nodes. Each node is interconnected with its neighbor through a duplex-link with 10 Gb/sec bandwidth, 6 µ delay, and drop policy based on drop packets from the tail of the queue. Each node contains 200 MB for the DaCy storage, i.e., network buffers, which results in a total ring capacity of 2 GB. These network characteristics comply with a cluster from our national super-computer center\(^4\), the target for the full functional system.

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

The first step in the experimentation is to validate correctness of the Data Cyclotron protocols using micro-benchmarks. We discuss three workload scenarios in detail. In the first scenario, we study the impact of the LOIT on the computation latency and throughput in a ring with limited capacity. In the second scenario, we test the robustness of the Data Cyclotron against skewed workloads with hot-sets varying over time. In the third scenario, we demonstrate the Data Cyclotron behavior for non-uniform access patterns.

---

\(^2\)NS-2 was developed by UC Berkeley and is maintained by USC; cf., http://www.isi.edu/nsnam/ns/

\(^3\)From our experience a high number of packets per data chunk and the continuous flow between dozen of nodes for several cycles exposed overflows for the sequence number and ACKs

\(^4\)http://www.sara.nl/
4.3.1 Limited ring capacity

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

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

To analyze the impact of LOIT on the Data Cyclotron performance behavior, we repeat the experiment 11 times, increasing LOIT on all nodes from 0.1 to 1.1 \( \text{loit} \) in steps of 0.1. Between two runs, the ring buffers are cleaned, i.e., all data is unloaded to the local disk.

Figure 4.2 shows the Data Cyclotron throughput for each LOIT iteration, i.e., the cumulative number of computations finished over time. The line registered computations represents the cumulative number of computations fired to the ring over time.

The experiment shows that a low LOIT leads to a higher number of pending computations in the system. For \( \text{LOIT} = 0.1 \) at instant 40 seconds, only 8000 out of the 30000 registered computations are finished. However, for \( \text{LOIT} = 1.1 \) at the same instant, almost 25000 computations were finished. We observe that the computation throughput is monotonously increasing with the increase of \( \text{LOIT} \).

Data access latency is also affected by low LOIT values. The graph in Figure 4.3 shows the computation life time distribution (histogram) for three LOIT levels. The computation life time is its gross execution time, i.e., the time spent from its arrival in the system until it has finished.

The results show that a high \( \text{LOIT} \) leads to lower life time of a computation. For example, the \( \text{LOIT} = 0.1 \) has a peak in the number of computations resolved in less than 5 seconds, but then it has the remaining computations pending for at least 100 seconds.

The reason for these differences stems from the amount of data removed from the ring over the time and the data chunks size. The workload hot-set is bigger than the ring capacity which increases the competition for free space in the ring. Using a low \( \text{LOIT} \), the removal of the hot data chunks is delayed, i.e., the pending loads list at each

\(^5\text{This is the upper limit of the function.}\)
Figure 4.2: Query throughput for multiple LOIT levels.

Figure 4.3: Query life time for multiple LOIT levels.
node grows. Consequently, execution of computations that wait for the pending loads is delayed.

With the optimized loading process for pending data chunks and a low drop rate, the tendency is to leave the big ones for last (cf., Section 3.4.3).

"The loadAll() executes postponed data chunks loads, i.e., data chunks marked as pending in the third outcome of the request propagation algorithm. Every TC milliseconds, it starts the load for the oldest ones. If a data chunk does not fit in the data chunk queue, it tries the next one and so on until it fills up the queue. The leftovers stay for the next call. This type of load optimizes the buffer utilization."

Whenever the least interesting data chunk is dropped from the ring, the available slot can only be filled with a pending data chunk of at most the size of the dropped one. Consequently, the ring gets loaded with more and more small data chunks, decreasing the chance of loading big data chunks even further. Only once there are no more pending requests for small data chunks the ring slowly empties making room for the big data chunks waiting to be loaded.

The graphs in Figure 4.4 and Figure 4.5 identify the data chunk size trend over time. The correlation between the ring load in bytes (Figure 4.4) and the ring load in
The data chunks (Figure 4.5) shows the data chunk length in the hot-set over time. With a continuously overloaded ring and a reduction of the number of data chunks loaded, the graphs depict that the load of big data chunks is being postponed. Therefore, the computations waiting for these data chunks stay pending almost until the end. The delay gets more evident for low LOIT levels. It shows that a priority policy defined only on size is not robust enough to reduce the data access latency. Instead, the priority policy should be based on their size and age.

The use of such a priority policy reduces the latency, but it does not improve the throughput in all situations. The reason is the absence of knowledge on how relevant the data chunk is for the throughput. Hence, the relevance of the data chunk for the hot-set should also be considered in the priority definition.

The experiment confirmed our intuition that the LOIT should not be static. It should dynamically adapt using the local DaCy storage load as a reference. In the next experiment we show the LOIT dynamic behavior when the hot-set is constantly changing and how well it exploits the ring resources.

Figure 4.5: Ring load in chunks.
4.3.2 Skewed workloads

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

Each workload $SW_i$ can enter the Data Cyclotron at a different time. In some cases they meet in the system, in other cases they initialize after the completion of the previous ones. This unpredictable initialization requires a dynamic and fast reaction by the Data Cyclotron. If $SW_j$ enters the ring while $SW_i$ is still in execution, the Data Cyclotron needs to arbitrate shared resources between the $DHi$ and $DHi_j$. It must remove $DHi$ data chunks with low $LOI$ to make room for the new $DHi_j$ data in order to maintain a high throughput. However, the data chunks from $DHi$ to finish $SW_i$ computations must remain in the ring to ensure low query response time.

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

The scenario created has four workloads ($SW1$, $SW2$, $SW3$, and $SW4$). Each $SW_i$ uniformly accesses a subset of the database ($Di$). Each $Di$ has a disjoint subset $DHi$, i.e., $DHi$ is not in $Dj$, $Dk$, $Dl$, with the exception for $DH4$ which is contained in $DH1$. Each $Di$ is composed of data chunks for which the modulo of their $id$ and a skewed value is equal to zero. The time overlap percentage between the $SW1$ and $SW2$ is 50%, 25% for $SW2$ and $SW3$, and no overlap for $SW3$ and $SW4$. Table 4.1 describes each workload.

From the previous experiment, we learned that the $LOIT$ should be inversely proportional to the buffer load. The dynamic adaption of the $LOIT$ is done using the local buffer load at each node. Every time the buffer load is above 80% of its capacity, the $LOIT$ is increased one level. On the other hand, if it drops below 40% of its capacity,

<table>
<thead>
<tr>
<th>workload</th>
<th>$SW1$</th>
<th>$SW2$</th>
<th>$SW3$</th>
<th>$SW4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>skewed</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>start(sec)</td>
<td>0</td>
<td>15</td>
<td>37.5</td>
<td>67.5</td>
</tr>
<tr>
<td>end(sec)</td>
<td>30</td>
<td>45</td>
<td>67.5</td>
<td>97.5</td>
</tr>
<tr>
<td>computations/sec</td>
<td>200</td>
<td>300</td>
<td>400</td>
<td>500</td>
</tr>
</tbody>
</table>

Table 4.1: Workload details.
the LOIT is decreased one level. For this experiment, we used three levels, 0.1, 0.6, 1.1 and each node independently adjusts its LOIT. The reason to only use three levels is to have the node’s LOIT aligned, this is, nodes will have similar adjustments to LOIT, and thus have better isolation to analyze the behavior of the ring as a whole.

Graph 4.6 shows the space used in the ring by each DHi. While, Graph 4.7 shows the amount of computations finished for each DHi.

The results illustrate the reactive behavior, i.e. how quickly the Data Cyclotron reacts to changes in the workload characteristics. The graph in Figure 4.7 shows a peak of 2000 finished DH2 computations between the 15th and 16th seconds. The graph 4.6 shows a peak in the load of DH2 data chunks in the same period. With the initialization of SW2 at the 15th second, the peak confirms the quick reaction time. The same phenomenon is visible for all other workloads.

The ring was loaded with data from DH2, however, the data from DH1 was not completely removed. It is a consequence of the 50% time overlapping between SW1 and SW2. In Graph 4.7 SW1 computations remain visible until the 43rd second. The data chunks to resolve these computations are kept around as it is shown in Graph 4.6. The Data Cyclotron does not remove all data from the previous workload in the presence of a new workload until all computations are finished. It shares the resources between both workloads as predicted. Observe that the sharing of ring resources gets lower as the time overlapping between SW decreases.

The SW3 workload shows an interesting reaction of the Data Cyclotron when it encounters a nearly empty ring. The DH3 started to be loaded and the ring is near to its limit. In all nodes the LOIT is at its maximum level to free space as much as possible. No more SW1 and SW2 computations exist in the system. Therefore, the last data chunks for DH1 and DH2 start to be removed from the ring. Their removal brings the ring load down to 37.5% of its capacity, below the 40% barrier defined for this experiment. With this load each node starts to set its LOIT to its minimum level, i.e, the data chunks are now staying longer in the ring.

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

The abundance of resources is over when the SW4 workload enters the scene. The ring becomes overloaded again, raising the LOIT to higher levels. Therefore, the DH3 data starts being evicted.
Figure 4.6: Ring load for skewed workload.

Figure 4.7: Computation throughput for skewed workload.
4.3.3 Non-uniform workloads

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

In the previous experiments, the study of the data chunk LOI focused on their age. The average number of copies per cycle, i.e., the ring interest within a cycle, was not included due to the uniform data chunk access pattern. Therefore, we initiated an experiment to stress it using a Gaussian data access distribution. The uniform distribution for data chunk size is retained. The workload scenario of Section 4.3.1 is used with the new data access distribution. The Gaussian distribution is centered around data chunk id 500 with a standard deviation of 50. All nodes use the same access distribution.

In this kind of workload, the Data Cyclotron keeps popular data chunks longer in the ring and exploits the remaining ring space for less popular data chunks. The workload distribution is represented by the gray curve in graph 4.8. The in vogue group is constituted by the data chunks with id between 350 and 600 which were copied more than 250 times. The data chunks on the edge of this group, the standard data chunks, have a lower rate of copies. The remaining ones, with less than 20 copies, are the unpopular data chunks.

The in vogue data chunks are heavily used by the computations, i.e., their LOI is
always at high levels. Therefore, they are kept longer in the ring. Their low load rate, pictured as black in graph 4.9, is explained by the Data Cyclotron cold down process. With an overloaded ring and the LOIT at its highest level, the Data Cyclotron removes data chunks to make room for new data. The first ones to be removed are the ones with low LOI, i.e., first the unpopular then the standard data chunks. For this reason the in vogue are the ones staying for longer periods as hot data chunks.

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

The low rate of requests, represented as black, for the in vogue data chunks contradicts the common believe that in vogue data chunks should be the ones with the highest rate of requests, thereby a high rate of loads. The reason stems from the request management in the Data Cyclotron runtime layer. A request is only removed if all of its computations have pinned it. Having a high number of computations entering the system, the probability for an in vogue request to be pinned for all interested computations is too low. As a consequence, the request stays longer in the node’s catalog and its load postponed because the data chunk is most of the time seen within a DaCy-cycle.

The experiment results show a good hot-set management for a Gaussian distribution. The high throughput is assured by keeping the in vogue data chunks in the ring as long as possible. For a low latency and large number of standard data chunks, the

![Figure 4.9: Gaussian workload: loads distribution.](image)
LOIT is used at its high level to reduce the access time to them. The in vogue data remains in the hot-set despite the high level of LOIT.

### 4.3.4 New level of interest

For a long number of DaCy-cycles, it was observed that an in vogue data chunk is exposed to the risk of being unloaded due to a drastic deviation on the ring interest on its latest cycle. In the presence of an outlier cycle, this is, sporadically the ring interest in the chunk drops or increases drastically, the hot-set management uses the data chunk’s history to absorb a drastic deviation. However, the weight of the data chunk’s history decreases over the time due to its division by the number of cycles (cf., Section 4.2).

\[
\text{newLOI} = \frac{\text{LOI}}{\text{cycles}} + \text{CAVG} \quad (4.3)
\]

To improve the absorption of drastic deviations LOI was modified to use the relative change between the number of copies between the actual cycle and the previous cycle. Furthermore, the history weight is not anymore directly dependent of data chunk’s age in the hot-set. Its weight is now controlled by a constant \(K\). The modification does not change the general behavior of the hot-set management observed in previous experiments. It only makes \(LOI\) calculation more robust against drastic deviations after a high number of DaCy-cycles.

The new formula uses data chunk properties; number of copies from the actual cycle (\(C_x\)) and previous cycle (\(C_{x-1}\)), number of hops (\(H\)), and the previous \(LOI\). The number of cycles is not used anymore. The \(newLOI\) is then calculated as follows:

\[
\text{RELCHG} = \frac{C_x}{C_{x-1}}
\]

\[
\text{CAVG} = \frac{C_{x-1} - (1 - \text{RELCHG})}{H} \quad (4.4)
\]

\[
new\text{LOI}(x) = \frac{\text{LOI}}{K} + \text{CAVG}
\]

We first determine the relative change, \(RELCHG\), between the actual number of copies \(C_x\) and the previous number of copies \(C_{x-1}\). It normalizes \(C_x\) with \(C_{x-1}\) to avoid the cases where the absolute difference is too big. If the number of copies increased relative to the previous cycle, \(C_{x-1}\) is increased by \(RELCHG\), otherwise, decreased:

\[
C_{x-1} - (1 - RELCHG) \quad (4.5)
\]
The previous $\text{LOI}$ for a data chunk carries its ring’s interest history during previous cycles. The variable $K$ can be adjusted or modified at configuration time. It can be used to tune hot-set management to be more aggressive in using history to attenuate deviations. In the simulation we have used $K = 2$, this is, $\text{newLOI}$ contains half of the previous $\text{LOI}$ to propagate the ring interest history over the time.

A simple simulation of two data chunks flowing in a ring can show how the formula introduces a smooth fluctuation on their $\text{LOIs}$ for drastic deviations on the ring interest from cycle to cycle, i.e., the number of copies per cycle. Figure 4.10 pictures the correlation between their $\text{LOI}$ and the number of copies per cycle.

For the first ten cycles, the $\text{LOI}$ of data chunk one ($\text{CHK1}$) increased almost linearly despite the variations on the number of copies between cycle six and cycle ten. A clear absorption of drastic changes in the data chunk popularity is seen between the cycle fourteen and cycle twenty. The $\text{LOI}$, for both data chunks had a light deviation
despite the data chunks popularity inversion. Furthermore, it also shows how the data chunk’s history has a weight in the new LOI determination. Between cycle fourteen and nineteen, despite the drastic changes in the data chunk’s popularity, data chunk two (CHK2) only for a cycle had a bit higher LOI than CHK1.

Controlled by a constant, the data chunk history works as an optimistic adviser for the workload requests in the coming cycles. Data chunks that were popular in the last cycles might also be popular in the coming cycles and vice-versa. Hence, for situation like the one observed at cycle fifteen, a data chunk is not removed due to a momentarily tumble on its popularity.

$K$’s value is a configuration parameter. Higher is its value more sensitive becomes the hot-set to drastic deviations on the workload. During the experiment, it was visible that for $K = 2$ LOI had fast reaction to workload peaks and good absorption of outlier cycles. The fast reaction is important for workload scenarios similar to the ones studied in Section 4.3.2.

4.3.5 Conclusion

The three scenarios have shown how the LOIT needs to be adjusted and how it influences the throughput for skewed and non-uniform workloads. LOI was improved for a better control on the history propagation. Its new definition expanded capabilities of the hot-set management to be flexible for a bigger scope of workload scenarios. The model presented to manage the hot-set might not be optimal, but it is robust and behaves as desired. It is an open research challenge to find the optimal hot-set management. In the coming sections we propose further improvements and discuss how they are integrated to improve throughput and reduce response time.

4.4 Warm data

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 will then be classified as such. In case the loaded data 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 a 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, i.e., the data is in memory, but not in circulation.

A request, during its journey, collects an estimation of the interest on the data chunk. Once it reaches the data chunk owner, it triggers the load of the data chunk
from the cold-set to the warm-set. The estimation is then used to pre-calculate a possible LOI as if the chunk had been loaded. For each DaCy-cycle its LOI is updated. Once it reaches a value higher than the ones flowing in the ring, the data chunk is loaded to the hot-set, replacing a data chunk with lower LOI. In case of equality, it is given priority to the new data chunk.

The LOI calculation is thus a continuous process since the reception of the first request. When the data chunk is unloaded, the data chunk is not moved to the cold-set, but to the warm-set. Hence, the calculation continues with the reception of more requests. If hot again, the data chunk is sent back to circulation. The process ends once the probability to be used reaches zero, i.e., the data chunk becomes part of the cold-set.

In case of a low number of data requests to trigger the data chunk load, a time-out, defined as number of DaCy-cycles, forces the load to avoid data starvation. As it will be described in the coming Section 4.6, the load of this type of data chunks combined with dynamic ownership model has a low impact on the flow of data chunks with higher popularity.

4.5 Cache

A computation can be composed of a single instruction or a set of instructions which are executed in any order such as map phase in MapReduce. In this type of computations an instruction execution is only dependent on input data from the storage ring.

Nevertheless, it can also be composed of a set of instructions for which the execution is also dependent on other instructions output such as a query execution plan for the relational model. Two different computations requesting the same data chunk might issue the pin() calls in a different order due to inter-operator dependencies. In most of the cases, the last computation to issue the pin() call for a data chunk CHK might have just missed the opportunity to retrieve it from the DaCy buffers without requesting it. Therefore, the computation remains blocked until the data chunk passes by again or, in worst case scenario, be reloaded. In case of an unpopular data chunk, the reload will take storage ring resources from the popular ones.

As a result, the response time increases due to a tiny time difference between the data chunk’s forwarding time and the pin() execution time. Runtime caching emerges as a solution for this type of applications to reduce data access latency. However, caching the latest used ones is sufficient due to the high competition for resources. Hence, two caching policies, efficient for the Data Cyclotron context, are here presented. Beforehand cache for generic workloads and a cache policy for non-uniform workloads.
**Beforehand cache policy.**

The data chunks passing by are cached in case they have been requested but not yet pinned. However, not all of them can be cached due to space limitations. Hence, the ones with a \texttt{pin()} call being issued in the near future have high priority to be cached.

The runtime system through the registration time of the \texttt{request()} calls, and on previous \texttt{pin()} calls, estimates when the data chunk will be used\footnote{The estimation is more accurate if provided by the computation which is possible at registration time.}. Based on the estimated time, the runtime system predicts if a data chunk will be used before it completes another \textit{DaCy-cycle}. If not, and in the presence of enough resources, the data chunk is cached. The data access time is then resumed to a read from the local memory.

**Policy for non-uniform workloads.**

For large hot-sets and non-uniform workload scenarios as the one used in Section 4.3.3 caching the ones which will be used in the near future is not enough to improve response time. During experiments on Section 4.3.3, it was observed that the life time of \textit{unpopular} data was too short compared to \textit{standard} data chunks and even more compared to \textit{in vogue} data chunks. With the introduction of the warm-set in Section 4.4, the time to re-load them is longer than the time to re-load \textit{standard} data chunk. Hence, the \textit{beforehand} cache policy by itself is not enough to reduce response time.

The authors in Broadcast Disks \cite{broadcast_disks} had a similar observation. They observed the data access latency is reduced if the less popular pages are cached instead of the popular ones. For their pure pushed-based system, instead of using a standard page replacement policy which tries to replace the cache-resident page with the lowest probability of access, they propose a replacement strategy that replaces the cache-resident page having the lowest ratio between its probability of access ($P$) and its frequency of broadcast ($X$). This ratio is referred as \textit{PIX}.

The \textit{PIX} policy was demonstrated to be optimal under certain assumptions. However, it was not a practical policy to implement. It requires a perfect knowledge of the access probabilities and it has an expensive comparison to determine which pages should be evicted. Therefore, they designed and implemented \textit{LIX}, an approximation of \textit{PIX}.

In the Data Cyclotron, a pull-based system, there are two factors that make the implementation of \textit{PIX} policy feasible. The first factor is \textit{LOI} which shows the access probability. The second factor is the comparison granularity. In the Data Cyclotron we use data chunks of multiple Megabytes rather than Kilobytes pages. Hence, the deter-
mination of which data chunks should be evicted is trivial. By caching the less popular data chunks, the Data Cyclotron reduces the data access latency and the number of requests in circulation. At the same time, the in vogue data is not slowed down by the load and forwarding of unpopular data.

**Cache management.**

The cached data chunks are kept in the neutral zone of the DaCy storage (cf., Section 3.5.1).

"The DaCy sets a minimal number of buffer space for each type of data. The remaining space, neutral zone, is used for both data types depending on the workload requirements. For a workload with a small hot-set the neutral zone is used to cache data chunks for the application. Used data chunks are kept to be re-used by future computations. If they are under node’s ownership, they are ready to be forwarded up to request reducing data access latency."

The lack of space in this zone requires the drop of data chunks to free more space. The first data chunks to be evicted are the ones with the lowest $PIX$ followed by the ones with highest time to be executed.

The runtime cache is the first cache layer of the Data Cyclotron. The Data Cyclotron has another two layers, the hot-set and the warm-set. The organization of the cache layers in this three levels reduces the data access latency for pull-based systems which exploit a continuous stream of data. The trade offs of these policies and their evaluation are part of on going research, part of the study is presented with a full functional system in the coming Chapter 5.

### 4.6 Homogeneous hot-set

In all scenarios until now, each node contributed with more or less the same amount of data. Hence, the $LOIT$ had the tendency to be similar among all nodes. Some workloads might have a narrow scope of interest and mostly request data from a subgroup of nodes, to be called groovy nodes.

With very dissimilar DaCy storage loads the definition of $LOIT$ becomes imprecise. All nodes have the same amount of data passing by, but a different load rate. The ones with high load rate tend to take $LOIT$ to higher levels than the others. The step wise adjustment triggered by the pending loads is not sufficient to have only the most relevant data in the storage ring (cf., Section 4.2).
At each node, the runtime system derives the LOIT value using as reference the local DaCy storage load. Its adjustment is dynamic and inversely proportional to the DaCy storage load. If it is too loaded, LOIT increases. It is step wise increased until the pending local data chunks can start moving. If the data is being forwarded and loaded without delays it is decreased so the data is kept in rotation as long as possible.

The LOIT value should take as reference the minimum LOI observed in the hot-set. In case of a big discrepancy, such as three levels, it should be adjusted so the hot-set becomes more homogeneous. An homogeneous hot-set is composed only by the most relevant data. Like this, non-groovy nodes adjust their LOIT and remove unpopular data chunks independently if their local buffers are overloaded or not.

For the improved version of LOIT\textit{n} a node, per each DaCy-cycle, collects the LOI value of each data chunk passing by to determine the most frequently LOI value in the hot-set, i.e., the mode. The mode is used instead of the median, or mean, because the LOI distribution tends to be highly skewed. Its value and its standard deviation are used to adjust the LOIT\textit{n}. Since the mode value is not necessarily unique, and to maintain the node’s autonomous behavior, the LOIT\textit{n} is kept within an interval instead of being set with mode’s value.

Figure 4.11 has a LOI’s distribution example. Region A) denotes unpopular data chunks while region B) denotes popular ones. The in vogue ones are denoted by region C). Region B) is bounded by the standard deviation of mode. Hence, any node with the LOIT\textit{n} outside region B) is adjusted to be in the center of B).

For a non-groovy node the LOIT\textit{n} once set in the middle of region B) remains there due to the lack of tension on the buffers to load data. On the other hand, for a groovy node the LOIT\textit{n} has the tendency to be near the border between region B) and C) due to the tension on its buffers.

Request data ownership to load in vogue data chunks.

For overloaded rings and huge data sets the groovy nodes are under stress to load more data. When loading in vogue data, they have to wait for unpopular data chunks to complete their cycle and be unloaded by their owners. Hence, this waiting time is added to access latency of in vogue data. The situation becomes worst for large rings. To overcome the waiting time, they could sacrifice some of their standard data chunks, however, such decision would decrease throughput. Hence, the solution is to request the unpopular data ownership from remote nodes. Hence, a groovy node can unload a data chunk with a LOI significant lower than the ones it has to load. The data chunk
is thus moved to its warm set and the *groovy* node becomes responsible for its re-load into the *hot-set*. Mean while, the data chunk header continues its journey to inform its previous owner about the change of ownership.

A direct appliance of this ownership request is for data chunks loaded to avoid starvation (cf., Section 4.4).

"In case of low number of data requests to trigger the data chunk load, a time-out, defined as number of DaCy-cycles, forces the data chunk load to avoid data starvation."

Their ownership is requested by remote nodes before completing a cycle to release space for the flow of relevant data. However, they cannot be unloaded before they have reached the nodes who have requested their load, or in an extreme case, the first one.

The model brings a new level of flexibility and robustness against rough workloads. Once the data is distributed at the ring initialization, the data can then bounce from one side of the ring to the other side and be owned by any node. The data chunk’s ownership established at the data distribution time is not anymore lifelong.

At the same time, it is used to introduce the concept of speed lines, i.e., depending
on the data chunk relevance, its cycle time can be longer than a DaCy-cycle. An in vogue data chunk will complete a cycle within a DaCy-cycle while a data chunk with lower relevance might need two DaCy-cycles. The difference of speed in each line is equal to the probability of a data chunk be moved down from the hot-set to a warm-set during its journey.

4.7 Summary

The Chapter presents the hot-set management using LOIT as indicator how overloaded is the ring. The less popular chunks are identified through LOI and removed from circulation. In autonomous and dynamically way each node contributes with the best of its knowledge for an efficient definition of the hot-set.

A decentralized hot-set management can lead to an unfair management of less popular data when the storage ring is overloaded. To circumvent the problem a new state for the data was created as well as the integration of an innovative cache management. The cache management equips the Data Cyclotron with tools to reduce latency for applications where instructions have dependencies and cannot be eligible for execution based only on input data from the storage ring.

With the intention to create a system capable of scaling in the number of nodes and support workloads using huge data-sets, ownership request was introduced to have an homogeneous hot-set. Its integration contributes for a more precise definition of hot-set which is now refined to: a set composed of data chunks with the highest probability of utilization and optimal size for efficient network bandwidth usage, the highest throughput, and the lowest data access latency.

With this Chapter the Data Cyclotron foundation has been presented. The architecture (cf., Chapter 3) together with the hot-set management were conceptualized into a full functional system. The optimizations proposed in this Chapter and its integration with a DBMS are evaluated in the coming Chapter 5 using different hardware configurations and different workload scenarios.