Evaluation of Scheduling Policies in a Staged-Database System

[Course Project Report: Advances in Data Management Systems] *

Nicolas Bonvin
LSIR,EPFL
nicolas.bonvin@epfl.ch

Rammohan Narendula
LSIR,EPFL
rammohan.narendula@epfl.ch

Surender Reddy Yerva
LSIR,EPFL
surenderreddy.yerva@epfl.ch

ABSTRACT

Traditional database systems execute queries using *one query multiple operators* approach. Such systems do not cash on the common data or computation which could be used by multiple queries. Thus leading to poor performance. To overcome these deficiencies, Staged-DB approach has been proposed, where the philosophy is *one operator multiple queries*. A Staged-DB system splits a traditional DBMS into a number of stages and handles a stage independently. In this project we evaluate the performance of a Staged-DB system under different scheduling policies. Our study is limited to scheduling various components that constitute the *execution* stage of a Staged-DB. We evaluate different scheduling policies that fix the order in which various operators in the execution stage should be executed. We demonstrate the performance of the policies in terms of *response time*, *throughput*, and *memory consumption*.

General Terms

Database management systems

Keywords

databases, performance, scheduling, simulations, stages

1. INTRODUCTION

Query execution in database involves multiple steps. First the query is parsed and then the query optimizer produces an optimized query execution plan, based on the parsed tree and statistics available in the system catalog. This query execution plan is nothing but a tree of operators. In traditional database systems the execution engine based on the query execution plan loads and executes the corresponding operators. Here we see that a single query needs multiple operators. This kind of systems referred to as *one query multiple operators* perform well if there were only a single query in the system at a given time. For multiple queries in the system, each query would have its own query execution plan and thus its own list of operators. Here each query would be executed as if it was the only query in the system. In such multi-query scenarios some of the operators in the query plans could be overlapping, by which we mean they refer to the same data and produce similar results. The traditional database systems fail to exploit this commonality.

On contrary to "one query multiple operators" methodology, Staged-DB [2] proposes a new methodology referred to as *one operator multiple queries*. Here the whole system maintains the n-operators [3] unlike in the previous approach where each query maintains the required operators. The multiple query demands are queued into these n-operators. In effect an operator contains all the query requests, thus the operator is in a good position to detect if multiple queries need similar data. The gain is very significant if the common requests refer to disk related activities.

In many decision making systems the amount of overlap between the different queries is significant. In TPC-H benchmark queries, which is for decision systems, this overlap is indeed very significant.

In such Staged-DB systems, query execution component of the traditional DBMS constitutes a single stage. The current study aims at different ways of scheduling [1] the various operators in this stage. The rest of the paper is organized as follows- in Section 2, various scheduling alternatives are discussed. In Section 3, a detailed description of the system implementation is provided. In Section 4, the performance analysis is presented. We present the scope for the future work in Section 5 and we conclude in Section 6.

2. RELATED WORK

The Staged-DB concept was first proposed in [3]. The work considered the need for a Staged approach in DBMS systems. The work emphasises more on scheduling among stages of the system and did not consider in detail, the scheduling with in a single stage. Several scheduling policies for scheduling among stages were proposed in [2]. The proposed solutions were experimented on real-time DBMS. They did not consider the scheduling with in a stage, namely the Execution stage of the Staged-DB. In this project, we are dealing with this problem and first time, building a simulation environment for the purpose.

*A soft copy of this paper is available at `http://lsirpeople.epfl.ch/narendul/adb/adb.html`

3. SCHEDULING THEORY

3.1 Scheduling Policies

The staged-db execution model contains several μEngines, and each micro engine corresponds to an operator in a query execution plan. Every micro engine contains an input queue where the input request packets are placed.

The staged-db has several micro engines. On a single CPU, at any point of time, only one of the operators should be executing. The scheduler, based on the scheduling policy, decides which micro engine should be executing next.

In our current research project, we will be considering the following different kinds of scheduling policies

1. Round-Robin Scheduling
2. Heavy-Load First Scheduling
3. Light-Load First Scheduling

3.1.1 Round-Robin Scheduling

In this policy the system keeps track of available μEngines. This scheduler schedules one μEngine after another according to the order of engines present in the list. Each μEngine will be given fixed (pre-decided) amount of CPU slice. At the end of the CPU slice the μEngine will yield the CPU to the next μEngine.

3.1.2 Heavy-Load First Scheduling

Each μEngine has a queue for input requests. The load of a μEngine will be defined by the engine. This could be the number of packets in the μEngine's queue or also weighing the queue length by the cost involved in using the μEngine. So, every μEngine has a load factor which varies as the number of requests vary. Now the system has the information of all available μEngines and the corresponding loads. We sort all the engines based on the load and pick the engine which has the highest load and schedule it. The amount of CPU slice allocated to this chosen μEngine will be proportional to the engine's load.

Now we have two choices in choosing which engine should be scheduled next:

1. Sort the engines' list again and choose the heaviest load first. In this case, we are trying to give as much CPU as possible to the heavily loaded engine. However, there is a possibility of starvation for some operators in this model.
2. In Second choice, we can avoid this starvation by sorting the engines' list only when each of the operator has got CPU for some time. We sort all the engines in descending order based on the load and then round-robin through the all engines, when all the engines have got the CPU then we sort the engines again.

3.1.3 Light-Load First Scheduling

This scheduling policy is similar to the Heavy-Load First scheduling policy, however the only difference is we choose the lightest loaded μEngine for scheduling. The philosophy is- we are giving preference to a μEngine which can complete its work at the earliest. It is similar to Shortest Job First scheduling in the case of operating systems.

4. SYSTEM DESIGN AND IMPLEMENTATION

The staged database system is modeled using the *discrete event simulation* technique. All the computations, data needs, and dependencies are modeled using events in the system.

4.1 Design of the modeling

In the section, we deal with the details of the modeling. The system maintains a global system level event queue called `sysQ` . There is a system level scheduler called `sysSched` which schedules the events in the `sysQ`. Each event has a start time called `firingTime` and at this time the event said to be *fired* . The `sysSched` just schedules the events in the `sysQ` in the order of their `firingTime` - the event with earliest `eventStart` is fired before other events. A clock is assumed to be associated with the system to time the various timing requirements of the events. Each event has a certain computation to be performed which is explained in detail later.

The system modeling consists of the following components, each of which is implemented in `Java`. All the components are losely coupled and communicate with each using various events, as discussed later in this section.

4.2 Components of the system

The entire execution engine modeling is realized using the following system components.

1. *Global System Queue*- which maintains the `sysQ` described above.
2. *Dispatcher*- to initiate the first packet of the newly arrived query.
3. *Operator/ μEngine*- deals with all the logic related to an operator.
4. *Global Scheduler*- schedules various operators for execution.
5. *Main Memory*- keeps track of pages fetched from the disk and the result pages produced by the engines.
6. *Overlap Detector*- detects possible overlaps across queries for computation needs or disk fetches.

A detailed description of each of the components is given in the following.

4.2.1 Global System Queue

All the event related needs of the various system components are addressed by this queue. When an event fires, the responsible component dequeues the event and processes it. Based on the execution logic, it may insert a new event into the queue. The flow of various events and the logic is explained in detail in respective components. Each event always carries the following piece of data which aids for respective components' execution logic to proceed.

- `eventId`- to uniquely identify an event in the whole system. Through out the report, a sample event is referred as $Q_1.J_1.1$ which is interpreted as the first event for J_1 node of query Q_1.
- `componentId`- the component id for which it is addressed.
- `functionId`- this id helps the component to choose one of the functional logic blocks it consists of. For example, as discussed later in this section, a operator is divided into several functional blocks like *insert* and *execution*. This `functionId` helps the operator instance to choose one of the blocks to process with.
- `firingTime`- the Global System Queue processor (`sysSched`) picks the event at this specified time and invokes the corresponding component based on the `componentId` and the control is transferred to the component.
- `packet`- which contains various data and metadata that helps a component's functional logic to proceed. A packet contains the following information items
  1. `queryIdList`- list of queries associated with this packet. For example, for a packet destined for table scan operator, this list can be interpreted as the list of queries that requested that table scan.
  2. `queryPlans and queryNodes`- corresponding to the queries present in the `queryIdList`.
  3. `pageId`- information relating to page requests corresponding to disk pages or output pages of computations. All the data and computation granularities are modeled at page level as described later.
  4. `contextInfo/ statusInfo`- the component annotates the packet with some status information as to what extent the packet was processed so far. It can have a single flag to find whether this packet is already processed and a request to the child's output is sent or not.

The kind of events various components insert into this queue are explained in detail in the respective sub sections.

4.2.2 Dispatcher

Based on the query arrival pattern specific to certain simulation setting, this component schedules the next query. It picks the query plan and schedules the first event corresponding to the root node of the query plan. For instance, if the query plan has root node as a Join operator, it inserts event $Q_1.J_1.1$ into the Global System Queue. When the corresponding component picks up the event, it takes care of event generation for the rest of the query plan. Each query plan is in the form of a tree converted into string representation.

4.2.3 μEngine

This component forms the core of our DBMS system modeling. Each query visits a set of μEngines in its life time for various computation and data needs. The order depends on the policy of the current scheduler. A query's execution begins basically after the corresponding *first* event inserted by dispatcher is fired. The system model at the moment, supports the following μEngines.

1. `Join`
2. `Aggregation`
3. `Sort`
4. `WScan (Wait-and-Scan)`
5. `Scan`
6. `IndexScan`

Each μEngine is configured with the following parameters.

- `Unit CPU Time (UCT)`- which is assumed to be the amount of CPU time needed to finish *unit of work* by the corresponding μEngine. The unit of work for each μEngine varies from one engine to the other, which is explained further when a detailed description of individual μEngines is provided.
- `Unit Disk IO Time (UDT)`- which is the time needed to access the disk for the *unit of work* it performs during `UCT` amount of time.
- `Input Packet Queue`- holds various packets destined to the μEngine.

μEngine computations are modeled at page level. What it means is, in this model, a μEngine corresponding to a parent node in the query plan can execute when at least one page of each direct child are available in memory. This logic may vary depending on the internal functional logic of individual μEngines. With out loss of generality, this page level modeling can be assumed as tuple level modeling also. All the above μEngines are discussed in detail in the following.

1. `Join:` this Join μEngine basically implements the Nested Loop Join algorithm. With the page level modeling semantics, it waits for a single page of the *outer* relation and *all* pages of inner relation. Once the corresponding pages are ready in memory, it produces its output pages. The number of output pages is computed based on the `selectivity` parameter, which is an extra configuration parameter for this μEngine in addition to the general configuration parameters specified above. When a packet is processed, it either sends requests to the children, if the packet is processed for the first time, or updates the `statusInfo`. The `statusInfo` represents what child's pages are requested for, before and what are ready in the memory at the moment.

   The `Join` μEngine processes one single packet from the engine's input queue in one `UCT` amount of time. The `UDT` is set to zero for this engine, since it does not deal with disk accesses directly. It always accesses the disk via the `scan` engine.

2. `Aggregation:` The μEngine computes some kind of aggregate value (like sum, product) on all the pages of the child at once. When a packet is picked first time for processing, it sends a request packet to the child for all of its pages. Later on, it polls the memory for those pages. When all the child pages are ready, it consumes them and computes the aggregate in one shot.

   The `Aggregation` μEngine processes one single packet in `UCT` amount of cpu time and its `UDT` is set to zero.

3. `Sort:` This μEngine waits for all the pages of the child and consumes them in one shot. It is assumed that a sorted table with an equal number of pages is created and written to disk. So any parent μEngine in the query plan has to read that table to enjoy the output of the `sort` μEngine. The `WScan` μEngine is designed for this purpose.

   In `UCT` time, the μEngine, processes a single packet of the queue. If a request for this packet is already sent to the child, a check is performed for whether the corresponding output pages exist in memory or not. If yes, it creates the sort table. In `UDT`, it writes a single page of this sorted table to the disk. So the sorted output is ready after the size of the table's amount of `UDT` time units of the system time.

4. `WScan:` A `WScan`'s child is always an instance of `Sort` μEngine. When a packet is processed and the child (`Sort`) is requested, a randomly generated unique table name is sent to the child. When `Sort` finishes its execution, it creates a table with this table name. So the `WScan` polls for the existence of this table. When the table is available, it sends a request to the `Scan` μEngine to scan the table.

   In `UCT` amount of time, it processes a single packet and either schedules a request to the `Scan` μEngine or a `Sort` μEngine. The `UDT` is set to zero.

5. `Scan:` The `Scan` μEngine picks a packet and schedules disk reads for the table requested in the packet. In `UCT` amount of time, it can place a request for a single page fetch from disk. And this page appears in memory after `UDT` amount of time. Hence a table read needs number of pages in the table times of `UCT`+ `UDT`. The `Scan` engine uses `Linear Overlap Detector` for detecting overlaps in the input packet requests which is explained later.

6. `IndexScan:` this engine has the same semantics as of the `Scan` μEngine except for the overlap detection used is `Spike Overlap Detector` which is explained in later part of the report.

Each μEngine implements the following functional logic blocks.

1. `Insert-` A packet is *inserted* into the packet queue, when a parent node needs the node's output pages. The overlap detector is called for, at this moment to detect any possible overlapping.

2. `Execution Begin-` This block takes care of the whole execution logic of the μEngine. This block consumes the `UCT` and `UDT` units.

3. `Execution End-` At the end of the allocated cpu slice, this block is executed which inserts an event corresponding to the scheduler into the `sysQ`.

4.2.4 Global Scheduler

This component fixes the order of the execution of the various μEngines. It implements various scheduling policies described in the previous section. To start with, the `sysQ` contains a single instance of the Global Scheduler event. Later, the μEngine which is scheduled inserts the next event for the scheduler into the queue.

4.2.5 Main Memory

This component simulates the memory manager of the real-time DBMS system. It addresses the various memory page requirements of the other components. When a query generates an output corresponding to an input packet in the input packet queue, it inserts a new page into the memory. The page description carries the same information as the packet description. The module provides the following interfaces

- `putPage()`
- `pageExists()`
- `consumePage()`

The pages are *pinned* and *unpinned* as overlaps detected by the overlap detector and the pages consumed, respectively. When no pins exist on a page, the page is deleted from the memory.

4.2.6 Overlap Detector

This component tries to detect various overlaps in the packet requests and creates a single packet for all the overlapped queries. The system now captures overlaps only in the `Scan` and `IndexScan` μEngine packets. Overlap detection in other components is complicated and needs a detailed analysis of the subtrees to detect common subexpressions. Overlap detection works in two phases

1. **Overlaps in Memory-** when a packet contains a request for scanning a page of a table and the page is already present in the memory, the packet's request is not sent to the disk. Instead, the page in memory is pinned and the μEngine which requested this page scan sees the page in memory the same way as it was fetched by disk read. It consumes the page as usual.

2. **Overlaps in Input Queue-** when a packet arrives with a page scan and a packet with similar request is already waiting in the input queue, the packet is piggy backed onto the existing packet. So no new separate disk scan is scheduled for this packet. When the page is fetched into the memory, this piggy back information naturally carried over to the page description.

Two kinds of overlap detection mechanisms are supported by the system

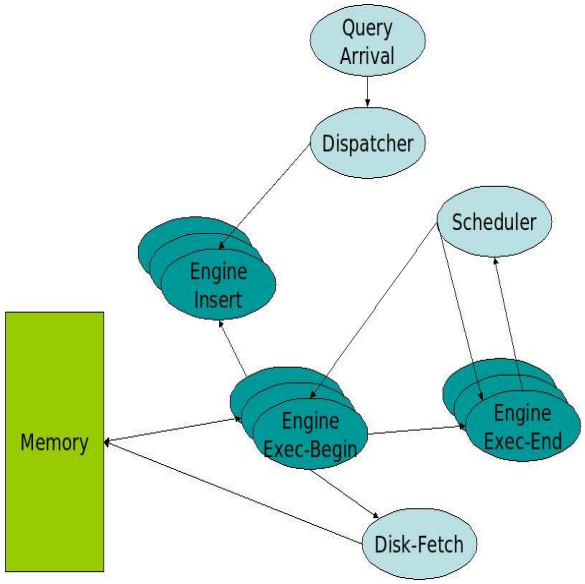

Figure 1: A high level overview of the system

1. **Linear Overlap Detector-** For normal unordered table scans, the page orders do not matter. Hence an incoming packet can be piggy backed onto any other existing page for the same table scan. So a late arriving query can make use of already existing pages in memory or yet-to-be-scheduled packets in the input queue. This is *linear* because the new query can arrive and exploit the old query at any time, of course, with a little loss of the overlapping. The `Scan` μEngine incorporates this overlap detection mechanism.

2. **Spike Overlap Detector-** For ordered table scans, index scans, the newly arriving query can overlap with existing query if and only if the first page of the table is still in memory or in the input queue. Here the overlapping window is limited to the first page itself, unlike the linear case. The `IndexScan` μEngine incorporates this overlap detection mechanism.

A high level overview of the query execution in our model is presented in Figure. 1.

5. PERFORMANCE STUDY

We performed a detailed performance study of all the scheduling algorithms.

5.1 Simulation model

There are two kinds of simulation possible as per the availability of memory resources is concerned. We can assume availability of memory is limited and reject all the incoming queries once we are out of memory. The other way is assuming availability of enough memory to satisfy all the incoming queries and observing which policy is consuming lesser memory. We adopt the latter in the model.

We have hard coded `10` sample query plans involving the said μEngines given in Section 3.2.3. The queries are selected with uniform distribution and introduced into the system with a fixed inter arrival time. We vary this inter arrival time and measure various performance metrics presented in next subsection. A maximum time limit on the system execution is set (50000 time units in this case) and we take the metrics at the end of this time. Each setting is simulated for 5 runs and each point in the graphs is an average of these 5 runs. We assumed the database consisting of 3 tables with number of pages 10, 15, 20 and the index scans are also available on the tables. The respective engines are configured with the following values in the simulation setting.

- For `Scan` μEngine, `UCT` = 1, `UDT` = 10
- For `IndexScan` μEngine, `UCT` = 1, `UDT` = 10
- For `WScan` μEngine, `UCT` = 5, `UDT` = 0
- For `Sort` μEngine, `UCT` = 15, `UDT` = 0
- For `Join` μEngine, `UCT` = 10, `UDT` = 0
- For `Aggregate` μEngine, `UCT` = 2, `UDT` = 0

5.2 Performance metrics

The following performance metrics were introduced to compare the performance of the policies.

- `Mean Response Time:` the time needed to produce the *first* page as the output. This metric rightly captures various interactive querying systems where the time for first output is important.
- `Throughput:` number of queries completed in a unit of time. The `Completion Time` of a query is the time needed to finish the query execution. It is measured as difference between the time at which the *last* output page is produced and the time when the query *arrived* into the system.
- `Maximum Memory Consumption:` the maximum number of pages consumed in memory during the life time of the query.

5.3 Results and analysis

5.3.1 Explanation of Legends

We have experimented with *Round Robin* scheduling policy in two modes- one where the slice is limited to processing just *one* packet (slice = `UCT`) and one where a fixed slice of 25 is given (slice = 25 `UCT`). The first mode is represented with *RR_0* and the latter with *RR_1*. Every policy is studied with both overlapping enabled (represented with a '1' in the legends) and disabled (represented with a '0'). The *Highest Load First* policy has a parameter referring to the percentage of the queue processed during its slice. *HL_50_0* refers to 50% case with overlapping disabled. The same applies to other legends. *LL* refers to the *Light Load First* policy.

5.3.2 Analysis of results

Figures 2-4 capture the performance variations of the policies with overlap detection enabled and disabled. Figure 5-7 demonstrate the performance only for overlapping case. The following observations are made from the graphs. The curves we are referring to are straightforward to locate on the graphs.

1. `Mean Response Time` of *RR_0_1* is much better compared to *RR_1_1*. This difference varies with a magnitude of 1.5 to 2. This is because in case of the former, the packets are processed very slowly. So the overlapping factor increases significantly as the new queries simply piggy back on existing packets. So many queries data needs are satisfied with a single packet processing.

2. `Mean Response Time` of *RR_0_0* is much more than *RR_1_0*. This is because only one packet is processed in a slice in the former case. So `Scan` μEngine can not satisfy page requests fast. In the latter case, more page requests are sent to disk in one slice and data needs are satisfied very fast.

3. **To summarize,** *when overlapping is enabled, we must delay processing packets in a graceful way so that more overlapping is possible. And more queries get benefitted by avoiding duplicate page fetches or computations. At the same time, this delaying should not be too high to make queries waiting for longer times. This makes a really an interesting future study.*

4. `Mean Response Time` of *HL_50_0* is almost same as for *HL_50_1* and growing linearly with the query arrival rate. It is because half of the queue is cleared in every slice and there is lesser opportunity for overlap detector to catch overlaps. Since the current μEngines do not have a local scheduling policy, packets which are waiting for their children output might be processed all the time and wasting the `UCT` units. As a result, the response time increases. But for *HL_100_0* case, all of the `Scan` μEngine's packets are also processed in one slice. Hence, results in faster page fetches which in turn, result in quicker execution of the queries. So the response time drops significantly and the curves are almost linear just above the X-axis. Here the *HL_100_1* case has slight advantage over *HL_100_0* when the query arrival is small. But later, there is no effect of overlapping.

5. **To summarize,** *for HL, the percentage configuration metric is crucial. We need to tune it, in such a way it does not hurt overlap detector and at the same time gets fine tuned response time. This study also reveals the need for a local scheduling policy. Only fresh packets should be processed compared to the packets waiting for the children. Current design did not consider this.*

6. The performance of *Light Load First* is observed to be the same as *High Load First* policy. It is because the number of μEngines we implemented are few. It will be interesting to study them in a better implementation environment. The curves are not presented in the plots for clarity reasons.

7. `Maximum Memory Consumption`- *RR_0_0* consumes more than *RR_1_0* because pages are kept in memory till the time they are consumed. In the latter case, more packets are picked for processing, hence more pages are consumed in one slice. This is even worse in the case of overlapping case. *RR_0_1* consumes even more compared to *RR_0_0*. It is because, in case of overlapping, pins still exist on pages unless all the queries which need them consume the pages, resulting in unpinning. *RR_1_1* consumes lesser than *RR_0_1* as more pages are consumed in the former case. *HL_50_1* consumes more than *HL_50_0* because of overlapping effect. *HL_100_1* less than *HL_50_1* because of more page consumptions in one slice.

8. `Throughput` *RR_0_0* has much lesser throughput than *RR_0_1*, because overlapping results in quicker page fetches for all the overlapped queries. With in overlapping case, *RR_0_1* has much lesser throughput than *RR_1_1*, because the latter processes the packets in big chunks in a single slice. *HL_100_0* has a lot better throughput over *HL_50_0* for same reasons. *HL_100_1* has much better throughput than *HL_100_0* for overlapping reasons.

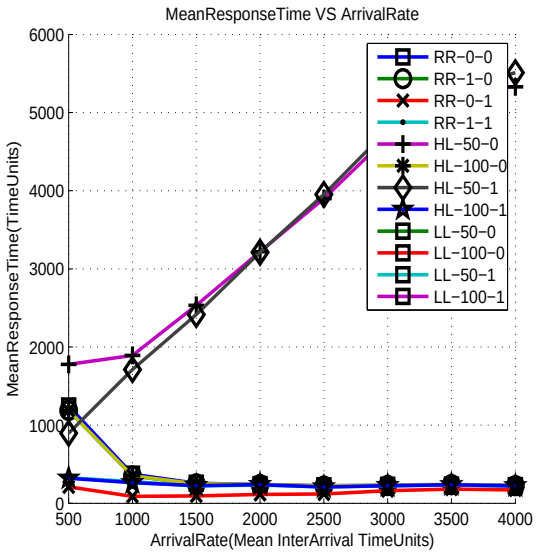

Figure 2: MeanResponseTime- with and without overlap detection- '0' for non-overlapping case and '1' for overlapping case

After clearly showing the merits of the overlapping case over non-overlapping case, we now try to study the performance of the policies with overlap enabled but different parameter configurations of the policies. Figures 5 to 7 demonstrate this. *HL*_10 is for the case when 10% of the input queue is processed when the control is given to the μEngine. *HL*_50 refers to the 50% case. Figure 5 shows that the RR policy seems to be performing the best. We need further experimentation to confirm this. The 50% case scores over when the query arrival rate is more over the 10% case for HL policy. Figure 6 reveals that the policies may do the same w.r.t memory consumption. We need further experimentation to confirm this. The same interpretation can be applied to Figure 7.

We learnt the following things in the observations

- The performance study clearly shows the need for intelligence into the μEngine which can be in the form

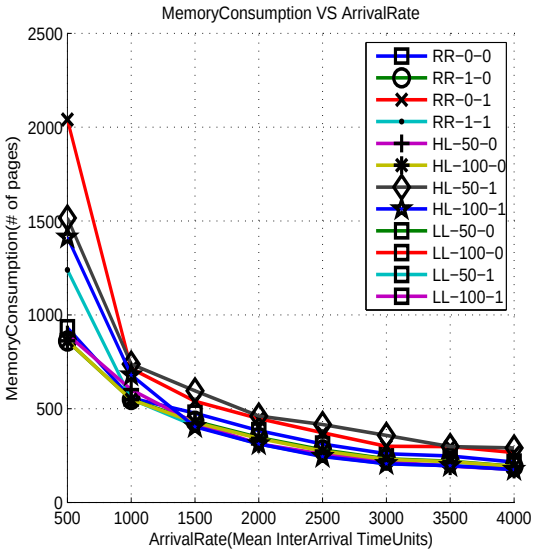

Figure 3: Memory consumption- with and without overlap detection- '0' for non-overlapping case and '1' for overlapping case

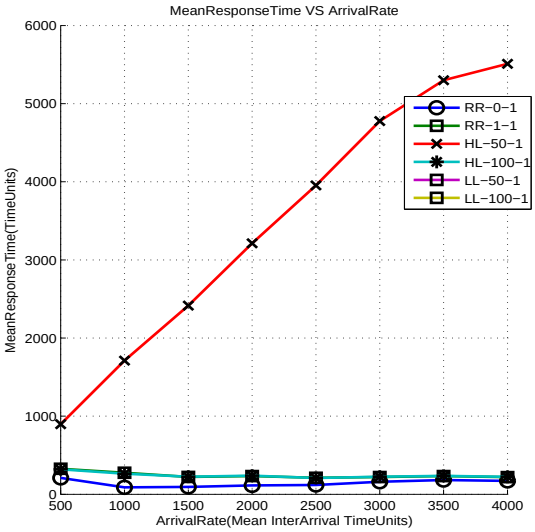

Figure 5: MeanResponseTime with overlap detector

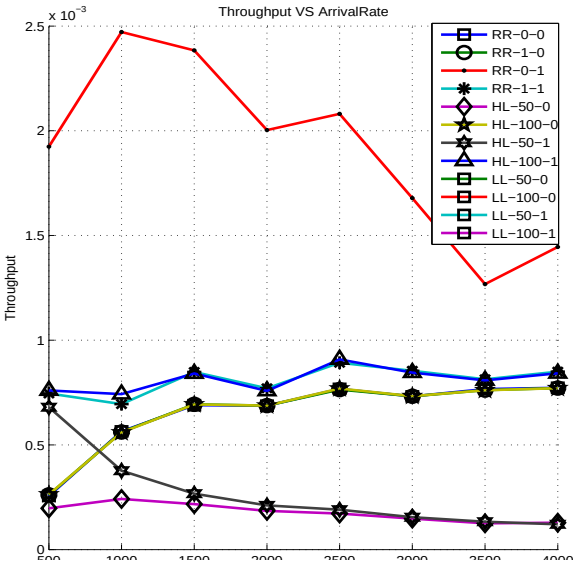

Figure 4: Throughput-with and without overlap detection- '0' for non-overlapping case and '1' for overlapping case

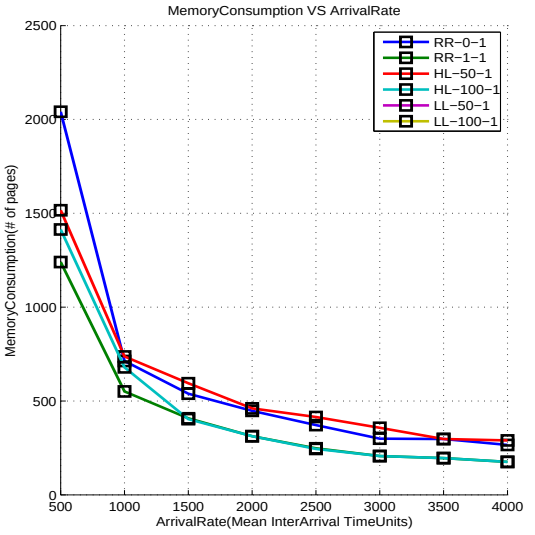

Figure 6: Memory consumption with overlap detector

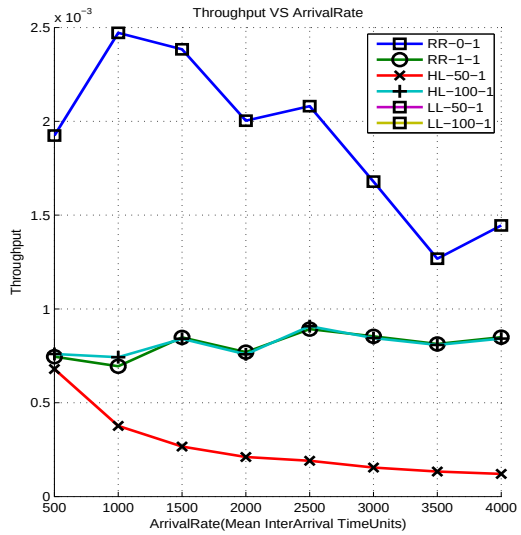

Figure 7: Throughput with overlap detector

of *local scheduler*. This scheduler should process more relevant packets in its slice than wasting the processing cycles on packets that are waiting for the children to finish.

- Every μEngine should tune its configuration parameters intelligently to cash the presence of overlapping. At the same time, it should be realistic and should not hamper the response times. More study needs to be done in this regard.

6. FUTURE WORK

We observe the following to be further worked out either with respect to the system implementation or design.

- Few more interesting global scheduling policies are possible.
- The system did not consider a *local scheduling policy* to pick one packet among many in the input packet queue, for processing next. It picks the fist packet in the queue at the moment.
- Regarding implementation, experimentation should be done with more μEngines and a bench mark style input queries.

7. CONCLUSIONS

In this research project, we tried to evaluate various scheduling policies for scheduling various operators with in the execution block of the DBMS. This implementation provides a nice platform and further experimentation can be done by extending various existing components or adding new components especially the operators.

8. ACKNOWLEDGMENTS

We thank Anastasia for the invaluable feedback and suggestions throughout the course. We would like to thank Ryan and Ipokratis for getting us started with the staged-db code and for providing us the query execution plans.

9. APPENDIX

Links to team members' websites

- Rammohan- http://lsirpeople.epfl.ch/narendul/adb/adb.html
- Surender- http://lsirpeople.epfl.ch/yerva
- Nicolas- http://lsirpeople.epfl.ch/nbonvin/courses/adms07

10. REFERENCES

[1] Harizopoulos and Ailamaki. Affinity scheduling in staged server architectures. In *Technical Report-March'02.*, 2002.

[2] Harizopoulos and Ailamaki. A case for staged database systems. CIDR'2003.

[3] V. S. Stavros Harizopoulos and A. Ailamaki. Qpipe: A simultaneously pipelined relational query engine. In *Sigmod'05*, 2005.