Persistent Back End for the ATLAS Information Service of Trigger and Data Acquisition (P BEAST)

by Alexandru Dan Sicoe

26.04.2011

Acknowledgments

The author would like to thank Giovanna Lehmann-Miotto (CERN) for supervising the project and for providing extensive guidance and support for the work done. Special thanks go to Luca Magnoni (Marie Curie ACEOLE fellow, CERN) for sharing his vast experience in database technologies and giving valuable technical advice.

Contents

Abstract

ATLAS is the largest of several experiments built along the Large Hadron Collider at CERN, Geneva. Its aim is to measure particle production when protons collide at a very high center of mass energy, thus reproducing the behavior of matter a few instants after the big bang. The detecting techniques used for this purpose are very sophisticated and the amount of digitized data created by the sensing elements requires a very large data acquisition system, based on thousands of interconnected computers.

The experiment is successfully taking data since the end of 2008 and the trigger and data acquisition are now in a production stage.The main development efforts are guided towards adding easy to use and intuitive tools to aid experts monitor different components or subsystems. P BEAST is an example of such a tool. It facilitates the storage of vast amounts of operational information which is otherwise lost. With this data at hand, long term analysis can be made and issues discovered. The project has reached deployment phase with promising results.

This report describes the work done and the results obtained in implementing a persistent system for the ATLAS Online Information Service. The structure carefully follows the evolution of the project. After setting the scene with a brief introduction to CERN and the ATLAS detector, an overview of the project planning and execution is given. Detailed descriptions of requirements, database technology evaluations, design, implementation and testing constitute the bulk of the main body. Achievements and future development plans are discussed in the last section.

How to read this report

This section is a short guide to the conventions used within the report for things like cross referencing, citations etc.

Citations are given by putting the index of the bibliography entry of a certain piece of work in square brackets after the text that refers to content of that piece of work. (e.g. **[5]**). If the report actually takes content from a source, then the citation will have next to it the pages of origin as well.

There is a bibliography section at the end of the report listing all literature surveyed. The notations of the IEEE referencing system were adopted.

Glossary

ACID Atomicity Consistency Isolation Durability

CF Column family

CLI Command Line Interface

CORBA Common Object Request Broker Architecture

CPU Central Processing Unit

CSV Comma Separated Value

EAV Entity Attribute Value

HLT High Level Trigger

IDL Interface Description Language

IPC Inter Process Communication

IS (Information Service) - subsystem of the DAQ Online system where applications publish information for monitoring and certain data exchange tasks

JVM Java Virtual Machine

LHC Large Hadron Collider

Partition - hardware/software infrastructure used for data acquisition at a certain moment in time.

P BEAST = **P**ersistent **B**ack-**E**nd for the **A**tlas information **S**ystem of **T**DAQ

POA Portable Object Adapter

PM Process Manager

ROD Read Output Data

Run (Run Control Run) the period of time when the run controller (IGUI) is in the 'Running' state and the detector is actively taking physics data. Not to be confused with an LHC Run which is a period in which LHC generates stable beams.

SA Shifter Assistant

SSH Secure Shell

TDAQ Trigger and Data Acquisition

TPS Transactions per second

Chapter 1

Introduction

1.1 Overview

1.1.1 Project goal

Design and implement a generic mechanism to store operational monitoring data of the ATLAS TDAQ system into a suitable database and provide an efficient means for retrieving that data.

1.1.2 Project motivation

Most of the operational monitoring data is transient and lost at the end of a data taking session. It would prove extremely useful for the data flow experts if they could analyze the quality of data taking a posteriori, accessing by means of specialized dashboards, information that is stored in a permanent location.

1.1.3 Background Information [1]

The ATLAS experiment

ATLAS is one of the four major detectors situated in the LHC tunnel at CERN. It is also the largest of them, with a height of 25 meters and a length of 45 meters. It is built from several layers of sensing elements (grouped in detectors and sub-detectors) each specialized in collecting data from specific particles emerging from a proton or heavy ion collision (an event). It also contains a toroidal magnet that exposes particles to a large magnetic field.

There are three main detectors inside ATLAS: muon spectrometer, calorimeter and inner detector. These can be further classified in sub-detectors and use different technologies to aid in measuring particle features like energy or trajectory in the magnetic field. For example, the inner detector has a sub-detector called SCT (Silicon Tracker) with sensors built out of very small silicon patches (micro strips). Each of these gives out an analogue or digital reading corresponding to the particle that went through it (or the number of particles). This makes the detector similar to a large cylindrical CCD camera that takes a snapshot of a short period of time after a collision has happened. They also allow the detector to locate, within a few microns, the region where a certain type of particle passed through. With this information physicists can later reconstruct very accurately the trajectories of particles in the magnetic field.

With the help of these sensors the detector outputs a total 140 million parallel channels of information to the underlying hardware (front end pipeline memories). From there information is passed by the detector read out drivers (RODs) to the first level of the data flow system called the read out system (ROS).

The architecture of the ATLAS TDAQ system [2]

The ATLAS TDAQ has two main roles. One is to find out when a notable collision has happened (trigger on an event) and the other is to assure the flow of data from the detector front end crates to permanent storage. The latter involves collecting and assembling all data from event fragments

(pertaining to the same event), filtering out the relevant ones (high level trigger - HLT) and transporting these through the system towards permanent storage so they can be later analyzed by physicists. The architecture of the system is split in 3 main levels:

Level 1 (L1): consists of hardware components that buffer and process event fragments from the fastest sub detectors (when the trigger is engaged). It computes what is the geographical region of these detectors containing interesting signals for that event (region of interest) and also filters the events reducing the input rate of 40MHz to about 75kHz.

Level 2 (L2): The 75kHz fragment data is sent to the readout system (ROS), a network of computers with customized PCI modules as buffers. Level 2 has a software system that only reads the event fragments in the ROS which are in the region of interest communicated by L1, processes them and makes a decision whether to reject the event or transfer it further, based on the partial information that it has. It accepts events at a rate of approximately 3kHz.

Level 3 (Event Filter): The event fragments from L2 are assembled into complete events by the sub farm inputs (SFIs). These full events are passed on to another software system called Event Filter which makes decisions whether to send them to permanent storage or discard them. Events typically have a size of 1.6MBytes and are archived with a rate of 200Hz.

On the whole, the TDAQ network acts as a pipeline that is controlled and monitored by around 30000 software processes (applications) running on several computer farms split in networks linked by Gigabit Ethernet lines.

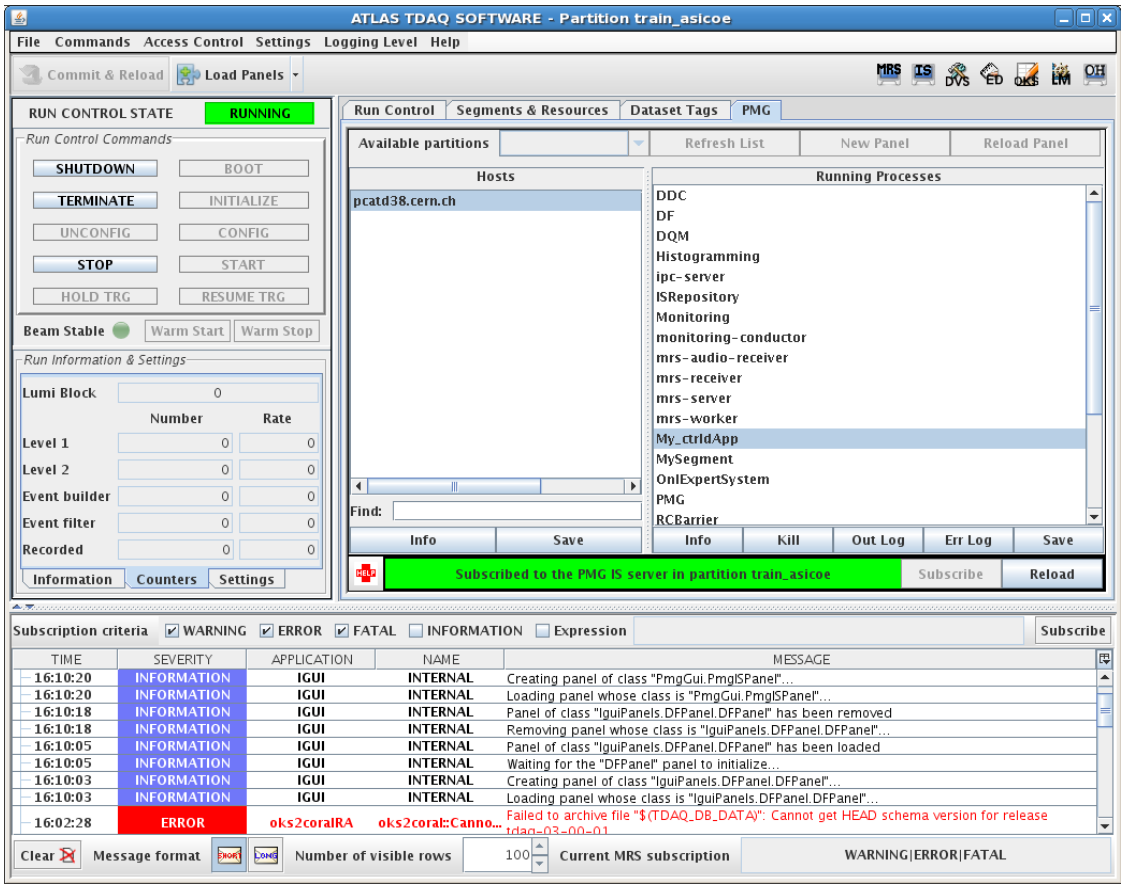


Figure 1.1: IGUI panel of partition train_sicoe in Running state

The control and configuration layer of the ATLAS TDAQ system

Is the software needed to configure, control and monitor the TDAQ system. It comprises itself of several layers of services:

User interfaces These allow users to visualize data generated by the different systems (e.g. IGUI in fig. 1.1 on the preceding page)

Artificial intelligence Expert System (looks at overall system and makes adjustments to keep things running smoothly), Shifter Assistant (makes suggestion to the human operator).

Core packages:

- Configuration Databases (OKS)

  When there are so many software applications running in parallel on such a large hardware infrastructure there has to be a way to ensure that everything is set up correctly. OKS is an xml database system that contains full description of partitions. A partition is a subset of the TDAQ hardware and software infrastructure which can acquire data. When ATLAS is taking physics data there is only one partition (called ATLAS). Other partitions can exist for calibration tests on sub detectors.

  An OKS xml file contains objects that can define anything from computers to applications. By editing this file, an administrator or developer can specify with great detail exactly what applications run on which machines and with what parameters. With the help of some of these parameters start-up dependencies can set between applications. To relieve administrators from manually editing files, OKS provides graphical user interfaces. It also has an API to programatically extract or alter configuration information (Data Access Library - DAL).

- The Run Control Skeleton/Tree

  Implements a finite state machine model, the states of which define certain stages of a data taking session of the ATLAS detector. While the completion of each state transition is automatic, the change of state is performed by human operators (run control shifters) by pressing buttons in the Run Control Commands panel of a specialized graphical user interface called IGUI (see fig. 1.1 on the previous page). In this way the ATLAS data taking is in full control of the operator whose job will be to bring the system in the Running state and keep it that way during a period in which the LHC provides stable beams (real collisions). This is of highest priority because it means that physics events are stored to be further processed by physicists on the GRID.

- Process Manager (PMG) and Resource Manager (RM)

- Access Manager - gives permissions to start/stop services

- Information sharing services

  - Event Monitoring Service
  - Online Histogramming Service
  - Error Reporting Service
  - Information Service (IS)

  These systems make use of a basic communication service called IPC (Inter Process Communication) which abstracts CORBA. Each of them accounts for a special type of information that can be shared among applications: physics events, histograms, error messages, operational information (status information).

  The operational information is published in the Information Service and is the one that P BEAST needs to archive. Currently a small fraction of this information is stored by a component called NetIS [3] but this has limited capabilities from the point of view of the generality and amount of the information it can gather. From here the need for a new system arises.

The Information Service (IS)

Represents a fundamental component of the ATLAS TDAQ and together with OKS is the main system with which P BEAST interacts. It consists of a multitude of server applications running on dedicated machines. They are used as a central repository for gathering useful information about all the other applications running in the TDAQ network of the detector. Each server instance can be configured at run time in either of: history keeping mode (keeps a certain number of old values in memory) or default mode (when a certain piece of information is updated its old value is lost).

The IS system has two main types of clients: information providers and information receivers. Providers can create new information objects in the IS repository or can update existing ones. Receivers may read object values or can subscribe for the changes of the repository to receive notifications when a specific object or set of objects are updated.

The information that is published in IS is vast and varied, reflecting any possible source from sub detectors and infrastructure applications to expert systems etc. IS uses an object model for representing information, which implies each piece of information in the repository has a specific type. Each type contains a set of attributes of primitive types directly mapped to C++ basic types (boolean, s8/u8, float, double, s16/u16, s32/u32, s64/u64, string) and arrays of these. Information objects of different types normally have different update intervals. Each time an object is updated the update time stamp is stored together with the object. This time stamp has microsecond precision.

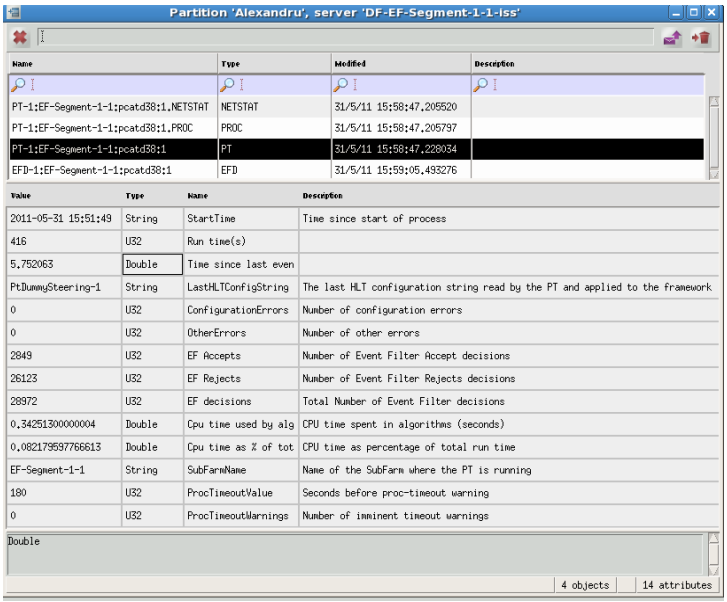


Figure 1.2: IS Monitor

There are many applications that can publish information of the same type. Generally they are identical or perform similar tasks, and will be grouped in the same IS server. The load is spread across multiple IS server instances (around 100 when ATLAS is recording physics data). In effect, any IS server is responsible for collecting information from a certain number of providers which can be of several types. For instance event filter applications publish information in IS servers that have a name starting with "DF-EF". A more concrete example of the structuring explained until now can be seen in figure 1.2 which shows a snapshot of the IS Monitor facility displaying the information published in the "DF-EF-Segment-1-1-iss" server running in a local partition. The information with name "PT-1:EF-Segment-1-1:pcatd38:1" is selected and its attributes are revealed. It is of type "PT" which means it is a processing task application which actually does the job of taking fully assembled physics events and deciding whether they will get sent to permanent storage (they are interesting) or are thrown away.

The communication between the IS repository and information receivers/providers is implemented on top of CORBA [25]. CORBA was used to simplify design and implementation of the distributed services of the TDAQ system. It does this by facilitating the communication of applications written in different programming languages without forcing the application developer to understand or even

know about the middle-ware that handles data representation, marshaling, remote method invocations, object reference integrity and naming. In fact, even CORBA details are abstracted by a general communication service called IPC. Like most of the TDAQ services, the IS implementation is available in both C++ and Java. In C++ the CORBA implementation called omniORB has been used, while for the Java applications the JacORB [26] broker has been chosen.

Some of the IS characteristics which were discovered to be important for P BEAST are listed below:

- Receiving an update for an information does not imply that all attributes changed value. This is because the IS servers see data that is published to them as a blob (they do not know the internals of the information that is published to them). If a certain information source publishes the same data twice, the IS will provide two separate update callbacks for that information.
- If an IS server process is stopped, no notification is given to its subscribers. This means that the subscription is lost and it has to be renewed by the client as soon as the server comes back up again.
- Updates are not guaranteed to be received in the order of their timestamps because IS is not meant to be a real time facility and due to the possible interleaving of multiple Jacorb threads.
- IS just sends the information to the subscriber objects via TCP/IP so as long as the request is received that the information arrived successfully at the CORBA client on the subscriber side, IS stops caring whether it was processed completely or not after that. However, if a request cannot be processed because the Jacorb buffers are full then the IS server times out and places the information that was not sent in a buffer. Then it spawns a low priority thread that gradually attempts to resend information from the buffer. However this mechanism can put pressure on the IS server if there are many slow subscribers and it should not be used as a reliable backup mechanism.

The list of IS servers that host the operational information that P BEAST needs to archive is summarized in the table in appendix 8.1 on page 51. It also contains a count of the total information objects published in each of them. The measurements were done during a typical run of the ATLAS experiment. 50 IS servers were found to host information of interest for P BEAST. They are of two categories: working IS servers (the ones starting with "DF-EF-Segment-","DF-L2-") and infrastructure IS servers (all the rest). The difference is in the lifetime of the IS servers: infrastructure IS servers are started when the partition is brought up and stay active irrespective of the run transitions while the others are dependent on the configuration of partition segments and come to life only in the INITIALIZE run control state. Apart from this, the rates at which they are capable of providing information are different (see next paragraph for an explanation of appendix plots 8.2 on page 52 which emphasize this fact). In total, there are 170729 information objects which get published (created, deleted but mostly updated) at different rates. A similar measurement on the IS servers generated in a smaller partition used for testing yields 3000 information objects.

Another interesting investigation into IS is regarding the typical rates it generates during detector operation. Such measures are useful for understanding the nature of the load that can be provided by IS to P BEAST and are best visualized by graphs of the number of updates generated per second (TPS) during detector operation. These were initially provided by an expert system called 'Shifter Assistant' (SA) which supervises data flow monitoring information published in IS as well as other sources in order to replace the need for a data flow shifter control desk. For instance, for the graph in fig. 1.3 on the next page, each data point represents the callbacks per second received from 32 IS servers (running in the real ATLAS partition) in one minute interval. The plot spans across several hours and fully covers a data taking session. It can be noticed that IS has a bursty behavior with peaks during start up or shut down transitions (RunCtrl server is capable of providing the highest peaks in information rate during these transitions). Similar graphs were used to analyze individual IS servers, in order to get a feel of how data rates are distributes among them. Looking at appendix 8.2 on page 52 it can be seen that infrastructure IS servers (fig 8.1 on page 52) have higher rates than the others (fig 8.2 on page 52). Note that these graphs show the count of callbacks in one minute and a scaling by 60 is required to obtain values in transactions per second.

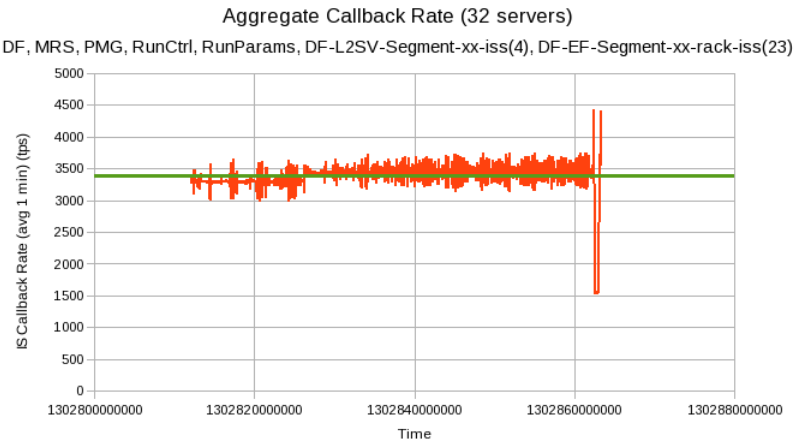


Figure 1.3: SA Aggregated IS Callback Count Per Min

1.2 Project planning

From the start of the project an iterative development cycle was envisaged to solve the problem in hand. The reasons for this choice of technique are: loose requirements, large scale of project, necessity to integrate with already existing complex systems, usage of new technology. All these amount to a large number of variables and trade-offs that are difficult to quantify or resolve beforehand. That is why it was considered that many of the technical issues would become clear as the project progressed.

Planning a precise execution timescale was found to be infeasible and it was replaced by an iterative development method. This approach was inspired from a known software development technique: agile development [6]. The project was split in several major phases: initiation, initial development, software improvements and additions and finally the project report write up. Each of these phases were further broken up into several tasks.

The initiation stage involved accommodation with the working environment and training (following a tutorial for developers with exercises intended to familiarize with TDAQ architecture, APIs of main components etc). In parallel, starting from a general specification agreed with the supervisor, a requirements document was to be written up and research work to be done in order to choose the most appropriate software platform. Supervisor and group review meetings were the main sources of feedback, advice and help with refining the requirements and reaching the optimal decision with respect to the database platform to be used.

The planning of this stage shows that initially the project followed the traditional software engineering approach with well defined analysis and requirements stages. However these were not totally separate but went on in parallel as was found necessary in order to avoid spending more time than needed in any one particular task.

The Initial Development and the Software Improvement stages were the ones that defined the agile nature of the project. The initial project Gantt Chart (See 8.6 on page 56) contains details with sub-tasks only for the Initial Development stage which was thought to span across several weeks because it was intended to be a full attempt at designing, implementing and testing the most basic functionality of the system. Not enough information existed however for a more in detail breakdown, such that all the subsequent iterations of the development cycle were encompassed in the Software Improvement and Additions stage.

The project report write up was left for the last period of the project when it was considered that a lot of the development would be already done and enough material would exist to put in the report. Also, by that time a more comprehensive view of the overall project could be given.

1.3 Methodology

The work done for the project is sectioned in prototypes which constitute the main trunk of development. This has several interaction points with the "customer" (the ATLAS TDAQ work group). Multiple channels of communication are used both to convey information about the project's progress and to obtain new requirements and feedback. For example, the research done on available database platforms and requirements is advertised to the group by maintaining a project TWiki page. Other on going activities are: periodic status presentations held during work group meetings, supervision sessions (concerned with the overall development) and short working sessions (discussions with various developers on specific technical issues).

This dynamic approach requires the use of appropriate tools to keep track of progress and important pieces of information. Apart from the already mentioned Gantt chart, the project working log is extensively used to make the daily task list, record issues, fixes and results. Separate logs are kept for meetings and user feedback.

In the initial project plan it was envisaged the project would have an Initial Development stage which would comprise of several sub stages: Initial Design, Initial Implementation, Initial Integration, Initial Testing. This was projected to last 45 days from 28 February – 29 April. This stage would correspond to a full blown prototype of the system. What actually happened was that this stage was broken up in multiple sections each of them still corresponding to prototypes of the full system but more limited in capability. The reasons for this are: the constraints and the performance requirements could not easily be quantified in very strict terms (e.g. average/max callback rate), the performance testing of the database platform was still in development (as well as experiencing its capabilities), there is the possibility of using different levels of testing infrastructure with respect to the scale of the partition (local machine, preseries cluster and ultimately the ATLAS detector infrastructure). Moreover, testing the platform within the actual ATLAS partition with real operational data from the detector infrastructure requires time. It would not make much sense to go through all that effort of deploying a young application directly there.

In essence, the project naturally followed a much lower granularity of the iterations than the one that was planned for. Instead of progressing in leaps equal in size to the initial development stage (which was seen as a full development cycle from design to deployment of the system on the full ATLAS infrastructure) much smaller steps needed to be taken. These were called prototypes and were assigned numbers starting from zero.

Each prototype corresponds to a set of major improvements added to the PBeast system. They are like small software projects, having a design, implementation, integration and testing stage. The importance of each of these stages within a certain prototype depended on how advanced the prototype was. For instance, the first prototype (0) was seen as an attempt to grasp the basics of the Java client API for Cassandra (Hector) and of seeing how and if data from the IS could be fit in the Cassandra database. That is why not much attention was paid to the design stage, the focus being on implementation. In prototype 1 however, the design stage was of highest priority mainly because the experience acquired from the previous prototype gave it the incentive to become the prototype that would offer an implementation much more closer to the top level functional diagram developed during the requirements stage.

The testing phases of the first two prototypes resumed to visual checks of the operation of PBeast within a partition generated on the local machine which allowed for a lot of the minor bug fixing. It did not make much sense to obtain quantifiable performance results at these early stages until the basic level of functionality was achieved and could be observed to function correctly, so rather, the effort went into design and implementation.

Prototype 2 however saw the addition of the final elementary feature, that of archiving only attributes that changed since the last value inserted in the database. Having achieved a system with a complete basic functionality, more emphasis was put on tests which were extended to a partition running on a cluster of computers (called "preseries"). Running the system at a larger scale allowed the making of important performance measurements and observations, useful for estimating the scale of the ATLAS partition and what improvements the system needs in order to cope with the real data.

The integration phases of the first three prototypes were minimal because other than retrieving information from some predefined IS servers (which was implicit via the IS API through the Jacorb implementation of CORBA) no other communication was made with any other components of the DAQ infrastructure. Each of the PBeast prototypes was started manually when the partition was in

the "Running" run control state. They first created the database schema, by doing an initial read of the information published in the IS servers and then they archived data received in the callbacks. Prototype 3 however was more geared to integrating with the partition infrastructure. PBeast was configured to be automatically started in the "Boot" stage and further functionality was added to deal with the dynamic behavior of IS servers (an issue only observed during testing of prototype 2 on the larger scale partition).

1.4 Risks

The biggest risk the project has from a technological point of view is choosing an unsuitable database platform. As mentioned earlier, this was ranked as a high priority task from the onset and it was assigned considerable time. Despite this, it soon became clear from the multitude of platforms available, the time constraints, complexity and the flexible requirements of the project that there would not be enough time to thoroughly investigate all the available platforms to be sure the optimal solution is chosen. In fact, the goal was to find a promising platform based on a set of well defined rules (some of which can also be found among the criteria for platform selection. See section 3.3.1 on page 20). These are meant to detect as early as possible the eventual choice of an unsuitable technology and thus lower the negative impact on the project. They are:

- choosing only open source platforms running on Linux
- basing decisions only on evidence given in official papers or benchmarks
- starting implementation only after proprietary benchmarks are performed
- using a modular design approach to decouple as much as possible the database specifics from the rest of the code of P BEAST. In case later it is discovered that there is a big problem with the platform, this would keep the costs of integrating another platform at a minimum.
- initial fast prototyping of the complete system with meticulous observations of any indication that the platform cannot possibly satisfy any of the core functional requirements

Another important technological risk factor is the possibility of loosing the work done due to things like hard disk failures. Development is carried out on a main work station but due to the distributed nature of the project, a lot of the work needs to be accomplished on remote machines as well. Considering these aspects, periodic back-ups are performed at the end of each working day. If external hosts are used, first the remote files (e.g. log files, configuration files) are centralized to the main workstation and then the whole data generated or updated within that working session is transferred to an external hard disk (the main back-up repository).

1.5 Milestones

1.5.1 Initial project plan

This section will go into more detail about the most important tasks of the initial project plan and why these were thought to be essential to the project evolution. As mentioned in section 1.2 on page 8 it was necessary that these tasks ran in parallel because the progress of each would influence the other. For example, the preparation of the requirements document can start as soon as the general project specification is discussed and extends throughout the entire initiation stage, feeding off important information acquired during the developer tutorial and the preparation of the platform evaluation document. (See Appendix 'Initial project Gantt Chart' Figure 8.6 on page 56)

Developer tutorial

The first objective that appears in the project Gantt chart is completion of the ATLAS TDAQ developer tutorial. It was important to carry out this work because it offers a basic understanding of the ATLAS TDAQ Online software infrastructure as a whole, as well as in depth knowledge necessary for developing

applications that interact with components of the system (in the case of P BEAST, the most interesting such systems are the Information Service and the OKS Configuration Databases).

The tutorial comprises of a set of exercises meant to give a complete walk through the major monitoring and configuration utilities of the ATLAS TDAQ Online software infrastructure such as: OKS Editor (for generating XML documents that describe a partition of the detector infrastructure), IGUI (main graphical user interface for controlling the data taking sessions of the detector) and the usage of the various installation scripts in order to run the most recent release of the TDAQ software on the work station. The exercises also give the opportunity to write programs that make use of the APIs of the major components of the Online system: IS (Information Service), MRS (Message Reporting System), Run Control Skeleton (the control tree used to propagate state changes) etc.

Requirements gathering

This task was necessary for several reasons: the requirements depicted from the project description were broad and needed to be expanded on, there was no previous system in place that offered to store all the operational data from the IS (there were only systems that offered persistence to slices of specialized information published in the IS, some of which containing a certain amount of operational data), the intention was that data stored in the system be accessed by as many users as possible. For details of the actual work done during this stage see chapter 2 on page 13

Database platform evaluation

The goal of the project is to store data published by different applications during the various stages of the life of a partition. The challenges of this are: handling the storage of massive amounts of data and offering a fast retrieval mechanism to access that data. These considerations made choosing a suitable storage platform a top priority. The task of preparing a "Platform evaluation document" starts early on in the project and extends until the end of the Initiation stage as does the requirements gathering.

1.5.2 Project execution

This section presents the important points along the project execution much in the same way as section 1.5.1 on the previous page did for the initial project plan. It however contains explanations regarding many other important tasks, sub-tasks and decisions which influenced the course of the project and which are a more detailed view of the application of the initially planned tasks or additions to those. A Gantt chart that represents the actual course of action taken to complete the project can be found in appendix 8.7. The initial development and software improvement stages are kept and are marked with a red box. The actual evolution of the developed prototypes are marked with a green box.

From the onset of the project several objectives were defined. The first one was to analyze the data of interest from IS (the target data). This was done during the requirements gathering stage and it involved talking to several system experts from the TDAQ group as well as familiarization with the tools used to visualize data flowing in IS (the IS Monitor panel of the IGUI of a locally generated partition, the Web IS web interface showing data published in the actual detector infrastructure)(See section 1.1.3 on page 7). In doing this, good understanding of the information structuring and naming was obtained. The work done helped in later devising ways of organizing data in the database.

Secondly, information regarding who would benefit from the data stored in the system (the users) and how they would access the data was needed. This has a big impact on the design of the database schema and it turned out to be not easy to obtain. The IS is a conglomerate of specialized data from different subsections of the ATLAS TDAQ infrastructure and depending on which slice of data is of interest there are different requirements on whether individual values need to be retrieved over time, or they are to be aggregated or correlated with other data and so are usually queried together with other specific data etc. Moreover, from discussions with DAQ experts, it was discovered that the main purpose for persisting IS data was for investigating the cause of failures in different parts of the infrastructure. This automatically implies that it is very difficult to predict fixed patterns of how data is usually retrieved since the decision to read certain data from different parts of the system would be dictated by the problems that occur. Ideally fine grained access to the time series of any individual value published in the IS servers should be available so it can be used if needed.

This lead to defining a very general query (retrieve all the values of this parameter in this time interval) which could be the basic building block of any of the more complex queries which were still to be extracted from the users. The database schemas of the P BEAST prototypes were thus developed with this basic query in mind, leaving it up for reading performance tests to dictate whether the more complex queries will be fast enough on the general schema or whether the schema needs to be adjusted to better fit newly emerging complex access patterns. This approach was later found to match well with the way data is usually modeled in the Cassandra key value store (See 3.4 on page 23).

The next logical objective was finding a suitable database platform. This involved establishing a clear set of criteria by which to compare candidates. After extensive research on possible technologies, starting from the ones used within CERN and ATLAS (Oracle, RRDTool etc) a considerable number of other promising platforms were found. After applying the criteria the list reduced to just eight (RRDTool, Graphite, OpenTSDB, KDB, HDF5, MongoDB, HBase and Cassandra). These were analyzed in more detail and a final short list of only three platforms emerged (HDF5, MongoDB and Cassandra). In the end all the investigations pointed out Cassandra as being the best fit for PBEAST's use cases. See details in section 3 on page 19.

The most important objective however was managing to collect all the data of interest from IS and to fit it in the database. This represents the core functionality that P BEAST offers. It was deliberately built into all the prototypes starting with the first one because it was thought to evolve as more insight was gained into the problem.

Chapter 2

Requirements

2.1 Context of P BEAST

The main component of the TDAQ with which P BEAST will interact is IS (see 1.1.3 on page 6) which is the data provider. P BEAST will make use of the subscription API of IS to subscribe to several IS servers within any given partition and get callbacks for the information published from different sources.

In turn, P BEAST will offer a general API that can be used by any application that wants to access data from the underlying database. There are three types of such applications that were identified:

- any custom program written by an expert that wants to look at certain parts of the stored data
- already existing specialized GUIs for visualizing specific data from IS
- ADAM web interface [4] - this is a system which provides a unique portal to access statistical information from the various sources of the TDAQ system. A driver implementing the general protocol of ADAM will retrieve data from P BEAST using the offered API. Examples of SQL-like queries that need to be supported are:

```
SELECT * FROM 'tableName' WHERE start=timestamp and
stop=timestamp

SELECT col1, col2, col3, ... FROM 'tableName' WHERE
start=timestamp and stop=timestamp
```

P BEAST will have to periodically transfer its old data into a long term storage. It will use the API offered by CERN's permanent storage system (CASTOR [3] at the moment of writing this document) to do so. Also, in case very old data is requested by the user, P BEAST will use the same API to get the data from the permanent storage and serve it to the user. Figure 2.1 on the following page is a graphical representation of the context.

2.2 General capabilities of P BEAST

P BEAST will be split in five functional subsystems (called components from now on). These are: the Info Gatherer, the IS Info Processor component, the Database Insertion component, the Database component and the Database Retrieval component.

The Info Gatherer shall be capable of subscribing to IS and receiving callbacks from information sources.

The Info Processor shall make necessary type conversions and filter information according to configured plug-ins (e.g. filter out duplicates, various implementations of smoothing algorithms etc). The filtered values will be sent to the database insertion components.

The Database Insertion shall take the filtered information from the IS Info Processor component passing it to the native storage methods of the database component.

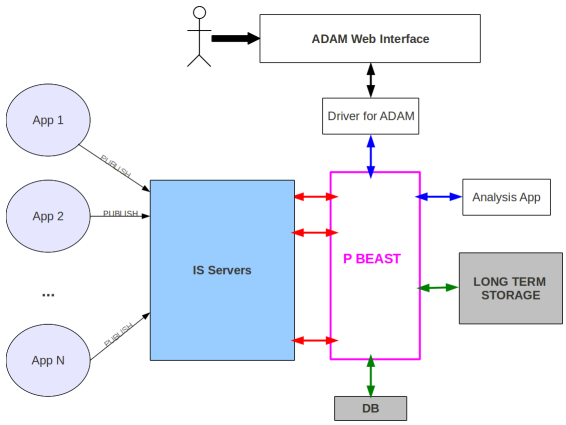


Figure 2.1: PBEAST Context

The Database Retrieval shall expose a common API for accessing information stored in the database component to any client application, masking the database technology used.

The Database is the technology that actually takes care of the storage and managing of the information. It offers the lowest level storage and retrieval API seen by the rest of the components of P BEAST.

2.3 General constraints on P BEAST

- P BEAST should run on the ATLAS Trigger/DAQ supported platform (at the moment Linux SLC5 [5]).
- Data providers are on an enclosed network Point 1 but the data has to be made accessible on the public network.

2.4 General assumptions and dependencies

2.4.1 Reliability

Assume that a 5% downtime is acceptable for the first version of the implementation and subsequent versions should improve this figure to a maximum 1% downtime.

2.4.2 Long term storage

Assume a storage system for long term data archival is available (at the moment CASTOR).

2.5 User characteristics

P BEAST will be used by any system expert, sub-detector expert or shifter who wants to have a more holistic view of certain interesting elements of the TDAQ data flow that took place in the past (days, weeks, months, years). Such a functionality would be useful for offline analysis: to understand short/long term past behavior of different parts of the system, to make comparisons between runs of the experiment, to investigate problems that occurred.

The user might also be a project manager (or any person working in the TDAQ group) who wants to use the data flow to serve as evidence of system operation in reports, presentations or papers.

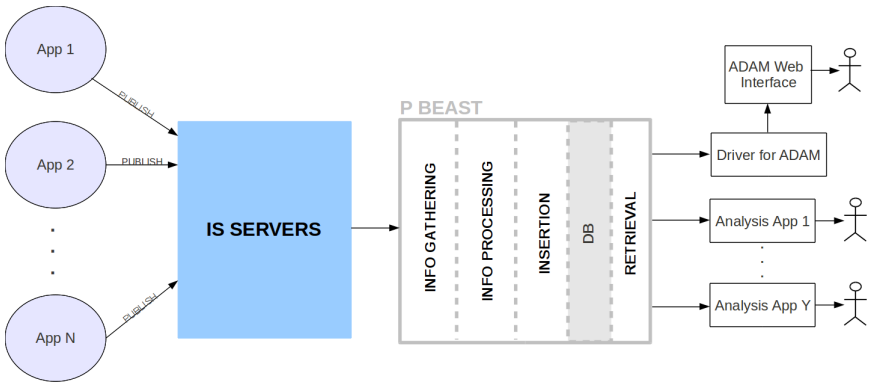


Figure 2.2: P BEAST Functional Components

2.6 Use Cases

2.6.1 Temporal access pattern

User looks at data for the last day

The majority of the times the user of the system will want to see just the most recent day of activity of certain areas of interest of the DAQ system. This happens especially when there is beam (detector is running). The frequency of this behavior is estimated to be 50%.

User looks at data for the last week

This is the second most frequently appearing use case where the user wants to see just the most recent week of activity of certain areas of interest of the DAQ system.

Note: The frequency of this behavior is estimated to be 40%.

User looks at data for the last month (or more)

The least frequent appearing use case where the user wants to see just the most recent months of activity of certain areas of interest of the DAQ system. The frequency of this behavior is estimated to be 10%.

2.6.2 Data access pattern

User wants to look at the whole data in a run of the experiment

In a period of data taking the experts most likely will want to access data for all the information sources from the experiment run when they recalled they had a problem (most likely previous run).

User wants to track single information across several runs

This is to compare the behavior of the same information published in IS between runs. It can be also useful for experts to figure out why the partition infrastructure behaved in a certain way in certain conditions.

2.7 Functional Requirements

These refer to the diagram in figure 2.2 which depicts the high level view of the system within its operating environment.

2.7.1 Information Gathering Component

Absorb IS data rates

Shall subscribe to all the operational information published in the IS servers of a given partition. The potential high input rate will most probably require to split the load of the callbacks from the IS servers across multiple sub components of P BEAST for better handling and responsiveness. Each such sub component will take data from an IS domain (subset of IS servers).

General

The system needs to be able to take in new types of information from new sources and dynamically insert them in the database – this is to accommodate for the dynamic configuration of the detector. Each information source can publish different types of information to IS. While most of these contain integers, doubles or status strings, there are other structures like variable strings, arrays of basic types etc. The average size of an information object is 100 bytes.

2.7.2 Information processing component

Filter the information

Binaries for different filtering algorithms (to be applied to the data) shall be set up when the system is configured.

These shall be attached to the IS clients at partition start-up and will be specified in the OKS configuration databases by a subsystem expert (See section 1.1.3 on page 5). Possible filtering algorithms:

- don't record duplicates
- record only fluctuations of a value above/below a certain value
- sample with a set fixed rate (smaller than the callback rate)

Configurable

The IS Clients component shall run within any DAQ partition and shall be configured via DAQ configuration databases by a system expert. Whether the filtering algorithms will apply to several IS servers (or individually per information source) will be configurable.

2.7.3 Database Insertion Component Functional Requirements

Take in information from IS Info Processor

The Database Insertion component shall be able to take in the data from the IS Info Processor and pass it to the Database component regardless of the deployments used.

2.7.4 Database Retrieval Component Functional Requirements

Expose common API

The database retrieval component shall expose an API by which any external application can access data from the database in a way that is completely independent of the underlying database technology used.

Use standard IS information source names

The API shall use the same information source naming format currently used in the IS (Partition.ISServer.Crate.Module)

Expose meta data

The API shall provide a means for the client application to retrieve the data schema at run-time. In order to become a permanent footprint of the operational information of IS, the database in P BEAST will have to have a flexible schema to be able to evolve in time as the configuration of the detector evolves (add remove sources of info) and as the data structures published in the IS evolve. In order for the retrieving applications to keep up with these dynamics P BEAST has to provide a similar functionality to that of the IS dictionary. This will imply suitable meta data will be stored in the database along with the actual information.

2.7.5 Database Component Functional Requirements

General

Shall be able to archive general data structures.

Handle concurrent data insertions and retrievals

The database technology used shall handle concurrent data insertions and retrievals.

2.8 Non-Functional Requirements

2.8.1 Project Deliverables Requirements

Design Document

A design document shall be provided at the end of the design stage.

Implementation Document

An implementation document shall be provided at the end of the implementation stage.

Users Guide Document

A user's guide shall be provided at the end of the implementation stage. This shall encompass important points like: how to configure IS clients, how to query the data etc.

Testing Document

A testing document shall be provided at the end of the testing stage. This will contain benchmarking suits for all the components of the system (IS clients, storage component, data retrieval component)

2.8.2 Performance Requirements

Gross input rate

The system may take in a gross input rate of 20 MBytes/sec i.e. approx 170 TBytes/year for typical yearly run times (estimation if all information was archived). This figure has to be taken as an absolute maximum information rate coming in the system. It was estimated (with the help of people directly involved with the data flow) based on the IS data currently taken in by the NetIS component (88000 variables/minute) multiplied by a factor of 1000 for typical yearly run times of 100 days/year. It is important to mention that this does not take into consideration the histogram information flowing, which is currently logged by the MDA (Monitoring Data Architecture). With an ongoing evaluation of the use cases of the system and the limits of relevancy for the information that needs to be archived, it is necessary to start from an absolute maximum upper bound to be sure that if it is later discovered that indeed all the IS information needs to be archived then the system was fundamentally conceived to accommodate that from the start. If at the beginning a lower rate is assumed and later a much higher rate is discovered in the actual system, then a fundamental redesign might have to take place which will be too costly. In order to get an idea of the storage space required the meta data (which can be a significant fraction) has also to be taken into account.

Write intensive

Shall sustain potential large data insertions.

Compactness

The overhead associated with the data should occupy a low amount of storage space.

2.8.3 Interface Requirements

Adaptability

Shall take into account the evolution of the data structures of the information published in the IS. What is meant by evolution e.g. might have a data class with an integer and a double and it might change to have two doubles instead. This is however a very rarely occurring process and it is not of high priority. Another example that would be more frequent would be change of the names of information (especially coming from sub detector applications) that is published.

Chapter 3

Storage platform evaluation

3.1 The big data environment

One of the key challenges of the project is choosing a storage system which offers the best fit to the requirements in Chapter 2. An incursion in the literature reveals a vast number of platforms available for storing large data sets. With the increasing growth of digital information flowing around the world [7] came a need for new technologies to manage vast amounts of data. There are two big areas that fueled this development: the commercial and leisure activities over the World Wide Web and the science of big data [8].

The first example refers to the world of companies offering commercial services to millions of users worldwide (Amazon, E-bay just naming a few). But this is only one of the business models existing in the digital world, other examples are big search engines (e.g Google), large free leisure portals (e.g. Youtube) or social networking sites (e.g. Facebook). These generate revenue by offering a free service, be it searching the Internet, hosting videos on-line or the possibility of interacting with people, which allows them to attract staggering amounts of traffic and then sell advert space to other commercial companies who want to expose their products or services to as many people as possible. All these new ways of interacting with digital data culminate with the advent of cloud computing which offers data and application hosting as a service.

The second example refers to big scientific domains like Earth studies, particle physics or biology where it is impossible to progress, generate new theories or verify them without analyzing vast experimental data.

In both cases mentioned above scalability of computing power and storage space is problematic. For example, anybody hosting an application on a social networking site could quickly find their one server not being able to sustain the number of requests if the application becomes popular overnight. In the highly interconnected digital space it is becoming more and more easy and even necessary to open up to potentially millions of users. To handle these highly increasing data rates and the variety of new use cases for data access, many new technologies have emerged in the last couple of years. The majority of these are part of a trend called NoSQL which brings together under one common name a series of extremely varied methods of storing and managing digital data which do not have much in common other than the fact that they are non-relational technologies.

3.2 Relational Database Management Systems vs Non-relational Systems

Apart from a comparison between the more than 30 year old relational databases and the newer ones, this section is a discussion about use cases. Relational technologies are well established, stable, with lots of experts having done lots of development using them. Their strong use case is activities where having safe transactions is of highest importance like banking systems (ACID - Atomicity Consistency Isolation Durability). In contrast, NoSQL encompasses all the other technologies such as: document storage, key value storage, distributed hash maps etc. What this newly emerging trend claims is that there are plenty of other use cases where the ACID properties become a burden for the performance,

scalability and maintenance of the increasingly growing data stores. Examples of such use cases are: time series monitoring data, geolocation sensitive applications etc.

The layout of the data affects the write and read performance of any storage system down to the level of the hardware. For the common mechanical hard disk, which is the most widely spread and has become significantly cheaper over time, random accesses (reads or writes) are slower than sequential accesses simply because for a random access the reading head needs to position itself on different sectors more often. This operation is called a disk seek and it is the longest operation performed by a HDD. This fact propagates up to file systems or DBMSs. It is however impossible for these systems to make sure that the data is arranged on disk in such a way that random accesses are kept to a minimum. One of the reasons for that is the big size of active data sets that applications need to access in comparison with what can fit in a disk sector, which means that random accesses will happen anyway and I/O operations are thus costly by default. That is why caches are used to avoid disk I/O: then the performance becomes proportional to the cache hit ratio. This technique is very widely used in relational databases where an initial schema is done which is set in stone during the application lifetime with eventual minor alterations. The initial schema can satisfy or perform well only for certain use cases defined by the developers at the beginning. But with time these will change and in order to accommodate new types of accesses, the flexibility of the relational paradigm is used (data is normalized) and increasingly complex queries are created, the results of which are cached in memory to assure performance.

In most of the members of the NoSQL trend however, data is denormalized (e.g. Cassandra). What this means is that data is duplicated in the form that would give the required retrieval queries. Instead of arranging the data in a fixed structure and modeling the queries based on what is needed, the data structure itself is modeled based on the users access needs and just some basic queries are provided by the platform to retrieve that data. This is based on the observation made in the previous paragraph that the cost of storage space has and is continuing to diminish and thus increasing storage space is not a problem. Basically, it means that it is cheaper to scale horizontally (having more machines with commodity hardware) than vertically (having performant machines with large and fast caches).

3.3 Research

3.3.1 Criteria for choosing the storage platform

Decisions have to be made considering:

- insertion and retrieval performance
- flexibility/generality of the data that can be handled
- availability as open source with strong community (active forums, mailing lists)
- optimizations for time series data
- amount of storage space needed
- usage in industry
- storage space scalability / durability / reliability
- maintenance

3.3.2 Methodology

Initially SQL technologies were considered. A simple exercise of checking how IS information would map to the relational model lead to the realization that the fixed SQL tables cannot provide the flexibility necessary for seamless evolution. In order for this to be attained using SQL, an Entity Attribute Value [11] method has to be used. This technique was previously applied at CERN with unsatisfactory results with respect to maintenance and the complexity/performance of retrieval queries, as the data set grew larger.

A whole range of other technologies were evaluated against the criteria. RRDTool, Graphite and OpenTSDB are specifically built for handling monitoring time series data but were ruled out for their

lack of generality (only support numerical values). KDB is a highly performant tool used for storing and processing financial time series data. Unfortunately it is a commercial product and just the 32 bit version is available openly. It was found too risky to rely on the company that develops it to keep it that way.

The strongest candidates that emerged were HDF5 [22], MongoDB [21], Casandra [14] and HBase [23]. They deliberately span across different technologies so that various storage solutions are explored, without relying on any one approach. Cassandra and HBase are both key value stores. Their common ground made it easier to compare them first, more in depth research remaining to be done for the rest (for a discussion of results see section 3.3.3).

3.3.3 Comparison

In comparing Cassandra and HBase it was found that the primary use case of Cassandra, high availability and write performance with specialized queries for retrieving slices of values, is a better match for P BEAST than that of HBase which is more focused on real time queries and performing computation across large data sets.

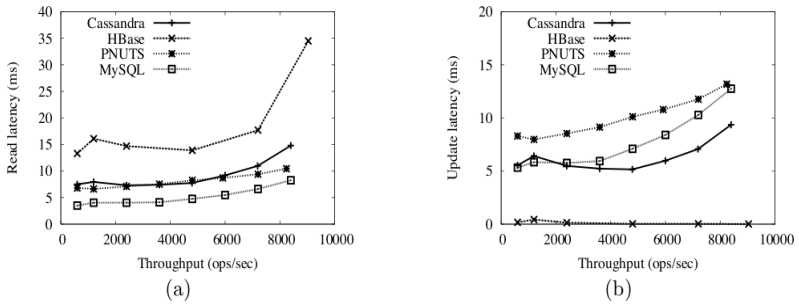


Figure 3.1: Workload: 95% reads / 5% writes - from [10] p.9

On top of this Yahoo! published a paper with benchmarking results for both Cassandra and HBase. These were done using their cloud serving benchmarking platform (see details about this in 3.4.7 on page 25). There are two plots of interest describing platform read and write latency for two workloads: a read intensive one and a more write intensive one. In both cases the primary interest is the write latency. In the read intensive workload (fig. 3.1) clearly HBase performs better but in the workload with more writes (fig. 3.2) the update latency of HBase, even though smaller than Cassandra's to start with, will begin degrading fast at a load of 8000 ops/sec while Cassandra's write latency is more stable across higher loads. This is a desired feature to have in order to satisfy the potentially growing loads of IS. The second workload is closer to the real usage of the system (an extremely write intensive one). For the read latency, Cassandra presents lower measurements in both scenarios. Thus Cassandra was favored to be analyzed further together with HDF5 and MongoDB.

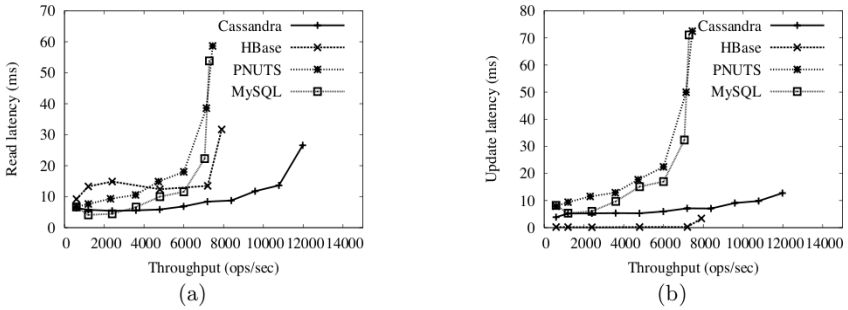


Figure 3.2: Workload: 50% reads / 50% writes - from [10] p.8

HDF5 is a file system with a library that provides functionality to access the files. It has a rich data set that supports multidimensional arrays. It provides powerful queries to manipulate in memory data which can be a subset of the data on disk. It is used by Matlab as a back-end and by Nasa's Earth Observing System for storing huge amounts of monitoring data collected by satellites.

MongoDB is a document-oriented storage system that provides very high schema flexibility. It supports queries similar in syntax to SQL to be made to the engine which retrieves one document at a time. It can be used as a very fast storage system for analytics data.

Cassandra is a key value storage system mapped on a distributed hash map. It is an Apache top level project used for storing petabytes of data by companies like Facebook (where the technology was initially developed). It has a strong use case for storing large amounts of time series data (used by Cloudkick [24] for monitoring cloud infrastructure).

	HDF5	Cassandra	MongoDB
Horizontally scalable	No built in functionality	Capacity can be added with no downtime	Automated sharding
Compression	Compression algs: -built in -easily add user defined	No server side compression.	No built in compression.
Durability	To some extent	Append writes to a commit log first	Does not keep an append-only log
Reliability	Not taken care in the platform	No single point of failure; Configurable	Asynchronous replication of data between servers for failover and redundancy.
Strong Points	Flexible (general) data model Strong well established technology	Massive write performance	Extremely flexible data model
Weak Points	Doesn't provide application to access the files	Possibly performs only with many nodes	Extremely poor durability Very large file size

Figure 3.3: HDF5 vs Cassandra vs MongoDB

Table 3.3 is a summary of the most important features of the three platforms. Based on this analysis the decision was made that Cassandra is the best suited technology for P BEAST (See section 3.4 on the next page for operation details and section 3.4.7 on page 25 for custom benchmarking results).

The main reasons that place Cassandra in better position than the others are:

- Cassandra offers the best fit to the use cases of P BEAST
  - lots of writes with fewer read operations
  - handles updates occurring with spikes [16] p26
  - designed to handle a very high write load (thousands of operations per second) [24] from multiple clients
- easy to increase performance by just adding more machines (cluster automatically reconfigures and re-balances itself)
- ease of deployment (facilitates the fast prototyping approach used for P BEAST)
- low maintenance requirements (important for the long term intended use)

3.4 Cassandra database technology

3.4.1 Cassandra technologies

Cassandra uses many new technologies. It was inspired from Amazon's Dynamo database from where it got the distributed ring arrangement and its focus on high availability (fig. 3.6 on the next page) and borrowed the storage data model and basic insertion and retrieval mechanisms from Google's Big Table (fig. 3.5 on the following page). It uses the SEDA [17] architecture for multithreading which allows more control on the computation process. To communicate with any Cassandra node there is an API on top of Apache's Thrift [18] protocol which is similar to CORBA [25], in the sense that it is based on an IDL and it allows remote procedural calls and facilitates communication between applications written in different languages.

3.4.2 Data model

The Cassandra data model is similar to a hash table that can have up to five levels. The first level is called a key space and is similar to the database in the relational world. Usually each application has one key space. Clients can access it by establishing a connection. The second level is analog to a table and is called a column family (CF). Each column family has a name and groups together entries which are sparsely associated with one another. The most basic piece of information that can be accessed is called a column. It consists of a tuple of three elements: name, value and time stamp. Since the time stamp is used for internal resolution of conflicts, a column is essentially a name value pair. Columns are organized in rows inside a CF. Each row has an associated key and stores columns in a predefined order of their names. The sort order is defined at the time a column family is created. This is done by choosing an appropriate comparator to fit the data type of the column names. This model offers two indexes for a CF, one based on the row keys and the other on the names of columns inside a row. This reflects in the queries offered by Cassandra (See section 3.4.5 on page 25). There is the possibility of a super-column family, which is a column family of column families which would add the last dimension.

Figure 3.4 shows two possible ways of structuring the data inside a column family. The gray squares represent columns (name value pairs) and the blue rectangles are rows inside a column family. The wide rows approach presumes the use of a CF with few rows but with many columns stored inside the rows (Cassandra documentation specifies that the architecture scales to billions of columns inside a single row). The other approach is to use many rows with a reasonable number of columns inside. As fig. 3.4 suggests, the lengths of the rows can vary because the columns are not predefined (like a relational table header). Each column can have any significance and can be placed in any row if desired. If the same effect is to be achieved in a relational table, null values would have to populate places in row which do not hold a value for a certain column name.

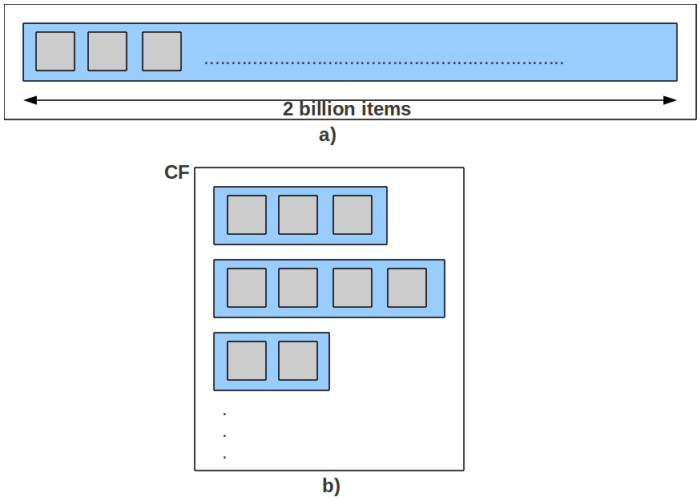


Figure 3.4: Wide rows vs skinny rows: a) Wide rows; b) Skinny rows

3.4.3 Insertion/retrieval mechanism

Figure 3.5 shows Cassandra's insertion and retrieval mechanisms, the architecture of which which closely matches that of Google's BigTable (See [13] p6). Before inserting a column, appropriate serializers have to be used for both name and value. Information is stored as arrays of bytes. This means any data type can be placed in Cassandra and can be read back correctly provided that an appropriate deserializer is used. The serialized values first get written to a commit log. This is a fast operation because it is sequential (values are written in the next available space one after the other, as they arrive) and no random write is performed. This assures that in case the machine goes down, the data that was in memory is not lost due to the fact that the commit log is replayed each time the server is started. From there, the data is immediately transferred to memory where it will reside in mem-tables. When the limit of a mem-table is reached (there are various configuration options to trigger this mechanism) data is transferred to disk in files called SSTables (Sorted String Tables [13] Sec 4, p3). These are periodically compacted (more SSTables are transformed into one) to save storage space. When a retrieval operation is performed, first a check is made to see if the data is in memory (in a mem-table or cache) and if not the indexes associated with the SSTables are searched to see if they contain the keys associated with the columns in the query. These point to the location in the file where the value for a particular column is stored.

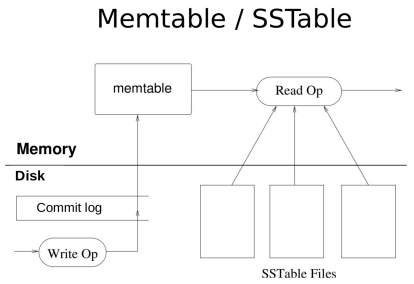


Figure 3.5: Cassandra read/write mechanism - taken from [15]

3.4.4 Cluster operation

All possible values that a key can hash to are arranged in a ring and portions of it are assigned to each node in the Cassandra cluster. The number of nodes where data from a certain node is replicated to is assigned by configuration. In figure 3.6 for instance, a replication factor of 2 is chosen, meaning that data stored in node B is replicated to the following two nodes in the ring, C and D.

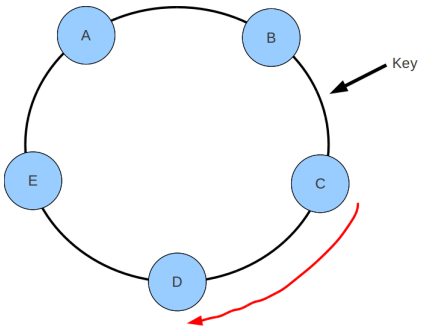


Figure 3.6: Cassandra DHT

Part of the reason why Cassandra is easily scalable and requires low maintenance is the fact that once it is discovered that the system's storage or computational capabilities are reached, a new node can be added to the ring. All the configuration work that needs to be done is to set its seed nodes in its configuration file and then the nodes will arrange themselves into the new structure. Automatic load balancing is obtained in the cluster by choice of a RandomPartitioner to distribute

the keys (based on a predefined hash function - MD5 [20] by default). The alternative is to use an OrderPreservingPartitioner which keeps row keys sorted by ASCII type and makes possible the execution of a query for a range of slices (which is not possible with the RandomPartitioner because the row keys are not stored in lexical order).

3.4.5 Retrieval queries

The Cassandra API supports retrieval of:

- A single column
- A slice (a series of columns from a single row). The columns to be returned are specified by: start name, end name, maximum number of results to return (range) or by giving an explicit list of column names.
- A range of slices (a range of columns from a range of rows).
- Multiple slices (a range of columns from a named list of rows)

3.4.6 Hector [19]

Is the most used Java API for Cassandra. It abstracts the Thrift API and implements features like a certain number of automatic retrials if an operation on the database fails. It offers the possibility of programatically creating a key space, a column family and it implements all the queries mentioned in the previous section.

3.4.7 Benchmarks

Yahoo Cloud Serving Benchmark (YCSB) framework

This is a framework released by Yahoo that can be customized to test different database platforms and how they respond to different workloads [10]. The workloads are themselves customizable and can be adjusted to fit the intended use case probabilistically (the distribution of query types i.e. percentage of insertion queries, percentage of retrieval queries; the probability of data already stored in the database to be accessed by the retrieval queries) and quantitatively (the amount of data to be initially stored in the database, the amount of load put on the database). This makes YCSB the perfect candidate to obtain some reliable performance results of Cassandra. Not to mention the fact that Yahoo! even published a paper where they presented extensive results where Cassandra was compared with several other database platforms including Hbase and MySQL. Those results place Cassandra in a good light for a certain read intensive workload (See figure 3.3.3 on page 21). This was encouraging but the workload that was required (generated by the IS) is an intensive write one with occasional retrieval queries performed by system experts via specialized (most likely web based) interfaces, when a problem occurs. From here the necessity to redo benchmarking on Cassandra, with workloads closer to the realistic use case, arose.

The interesting measurement to be recorded when performing the benchmarks was found to be read/write latency versus load on the Cassandra server. This is because the IS service has many instances that will provide callbacks whenever an information source creates a new piece of information (starts publishing it in the IS server), deletes an information (stops publishing an information in the IS server) or more importantly and frequently updates an information (republishes an information – most likely because at least one or more attributes of that information changed state i.e. something interesting happened, but this is not necessarily the case). The total rate of callbacks received from all the IS servers that P BEAST is subscribed to is the total maximum load that the Cassandra server can possibly be subject to at any one time. There is an entire discussion here to be made about the intermediate buffering that will be made in the part of P BEAST that gathers information from IS (subscribes to IS servers and gets callbacks and handles them). This buffer has the effect of lowering the instantaneous rate and keeping it steady. See more details in the design section 4.2.2 on page 32. Regardless of this, information will be coming into the PBEAST system at a given rate. The distribution of the IS rate over time is discussed in more detail in section 1.1.3 on page 6. In order for the system to function correctly and not loose any data it has to have enough time to process the

data and store it. To make sure that the storage mechanism does not stall the system, there are two things that have to be verified: whether the input rate of Cassandra is higher than the average callback rate of IS (this is the condition that has to be met to be absolutely certain that the system works in normal operating conditions) and whether the input rate of Cassandra can handle the maximum peaks in the output rate of the IS service (if this is true then the system should not have any problems with taking data in, if not then additional mechanism to absorb peaks have to be employed). In order to verify these two conditions before even starting development of PBEAST measurements of latency vs throughput to Cassandra are necessary. These measurements are discussed here. They have to be used together with measurements of the information service callback rates presented in the next section to make useful decisions about application design, implementation and deployment.

Several benchmarks using a customized version of the YCSB framework were performed. These were split in several Workloads: Workload 1 [10 cols per row, 1 million rows], Workload 2 [50.000 cols per row, 22000 rows], Workload 3 [100.000 cols per row, 50 rows], Workload 4 [500.000 cols per row, 20 rows], Workload 5 [1000.000 cols per row, 20 rows].

The benchmarking procedure is as follows: the initial workload is put in the database and then successive loads are presented to the Cassandra cluster employed. The objective is to record the write and read latency for 1 value and plot it against the load (measured by the YCSB framework in ops/sec). These queries that are part of the load are 80% writes (of 1 value) and 20% reads (of 1 value). In some cases slice reads of 50000 values were performed as well. These were chosen to better fit the realistic scenario of database use (heavy writes) although even a much higher write percentage could have been chosen. A 95% to 5% write/read ratio would have been maybe more closer to reality but then not enough information would be obtained to get a compelling view of the read latency. Obviously the data stored increases as measurements of latency (load) are taken because values are added on top of the initially stored data.

Before seeing some relevant results, it is worth mentioning the fact that Cassandra's value caching mechanism was not activated during benchmarking, but the row key caching was (it saves on average one disk I/O per read operation). Other than this, the default server configuration is used. Further optimizations can speed up the system like separate drives for commit log and data (See 6.5.1 on page 48 for an example of usage).

The result for Workload 2 in both configuration scenarios of the Cassandra server (running on one node or on three nodes) are discussed here because these were later found to be closest to the actual choice for partitioning of data.

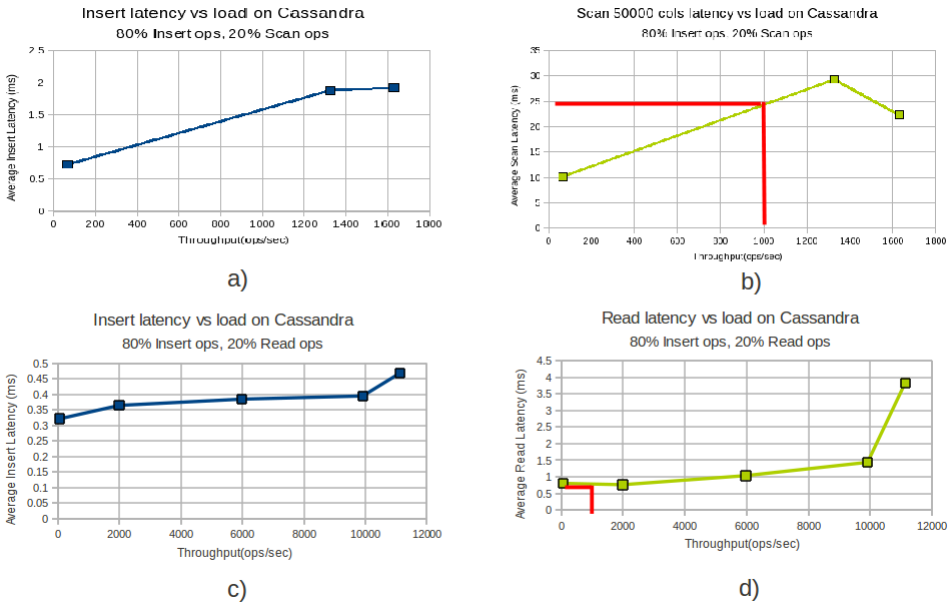


Figure 3.7: Workload 2: 1 node Cassandra cluster

Plot b) in 3.7 shows the latency of reading one value while plot d) shows the latency of reading

50.000 values. For a load of 1000 ops/sec the latency of retrieving 50.000 values is 25ms and the latency of retrieving 1 value is 1ms. Cassandra is optimized for retrieving multiple values (slice queries) and as can be seen the latency of retrieving 50.000 values is much smaller than the latency of retrieving one value multiplied by 50.000. This is also because 50000 network round trip times do not have to be done. Batch operations should thus be the preferred option for both reading and writing.

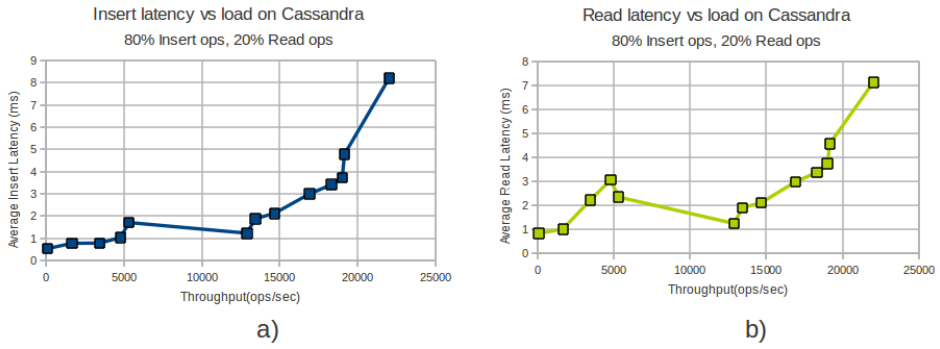


Figure 3.8: Workload 2: 3 node Cassandra cluster

By comparing plots c) and d) in 3.7 on the preceding page and plots a) and b) in 3.8 it can be seen that the performance of a 1 node cluster starts to significantly degrade after a load of 10.000 ops/sec while that of a 3 node cluster after 20.000 ops/sec. The latency has increased for the 3 node cluster due to the fact that operations need to spread across several machines. If a node receives a read query for data it does not posses it acts as a proxy routing data from the node that has got the data to the requester. Consequently, a write query has to be assigned to the node holding the range where the data key hashes to.

Another observation that was made during testing was the fact that the query latency is increasing and the processor on the Cassandra machine idles most of the time (the "top -H" [36] Linux command was used for checking). This is a clear sign that the system is spending most of the time waiting for disk I/O (it is hitting the I/O limit for the disks on the machine). A further incursion using the "iostat -x" command [37] confirmed this by giving the following output:

```
Device: rrqm/s wrqm/s r/s w/s rsec/s wsec/s avgrq-sz avgqu-sz await svctm %util
sda1 0.02 7.46 0.00 0.14 0.54 15.21 107.75 1.04 258.02 6839.44 100.00
```

The way to interpret this is that the sda1 hard-drive is the bottleneck on the server machine because it is completely saturated (100.00 in the "%util" column) and the average time it takes to enqueue and satisfy a request is 258.02 ms ("await" column). This is a very large time comparing to a single seek that typically takes between 5 and 10 ms.

Chapter 4

Design

This chapter describes the initial top level design of P BEAST and how this reflects in the design stages for each particular prototype. Detailed class and sequence diagrams are only provided for the design of the last prototype. This is the most complete version and encompasses ideas and features from the design of all previous versions.

4.1 Design levels

4.1.1 Top level design of P BEAST

The top level design emerged during the requirements gathering and database platform evaluation stages. Its main goal was to decompose the system into clearly defined functional blocks. These are in close relation to the functional requirements 2.7 on page 15. The obvious benefits of this approach are: the system is modular and the work that the system needs to do is split up in smaller, more easily manageable and independent blocks.

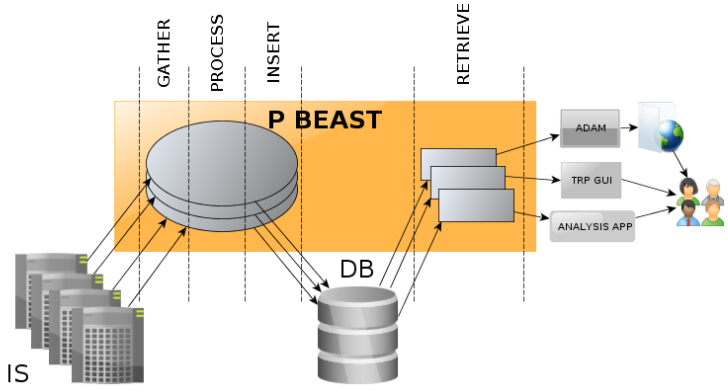


Figure 4.1: Top level design diagram

Figure 4.1 shows the initial top level system design and the interactions with other components in the on-line TDAQ system. P BEAST is composed of several modules which are meant to be decoupled. At any one point there are several instances of the component that interacts with IS, each taking data from a configurable set of IS servers. All of them will process data (received in an IS callback) and send it to the storage system. This is done via a component which offers the insertion functionality masking the underlying database technology used. On the retrieval side a simple programmatic API is exposed to different user applications.

4.1.2 Database schema design

The choice of database schema directly influences the retrieval performance. This is especially true for Cassandra (see 3.2 on page 20) and since this became the platform of choice, some of its characteristics needed to be taken into account in the design phase. Insertion performance can also be affected. This happens due to the dynamic behavior that the storage mechanism of P BEAST needs to have. New sources of information appear and disappear. In order to capture their data the system needs to make sure it has a place where to put that data. This implies dynamic schema creation which is an expensive operation and, if not handled correctly, it leads to missing certain windows of information and scalability problems. Moreover, in order to assure the correct evolution with IS, additional meta data has to be stored. This has to be simple enough and kept to a minimum to reduce storage overhead. In consequence, there are two separate types of containers that need to be present in the database schema: tables for IS data and tables for meta data.

The first category of tables (column families in Cassandra) evolved several times on paper, according to requirements and performance/functionality tests. In practice there is however only one major change in schema. Initially, in order to keep things simple the most natural layout for the data in IS was chosen: one column family per information name. This stayed implemented in the first three prototypes. The performance tests done in preseries on Prototype 2 however discovered that schema build times become prohibitive when many column families have to be created. Using one column family for every information source that publishes data in the IS servers of interest, rises the number of CFs to thousands. For the ATLAS partition they would be hundreds of thousands. Measurements found that it takes on average between 0.5 and 1 sec to create a column family. The realization of this phenomenon lead to a reevaluation of the way that the database schema is laid out. Four options arose (See fig. 4.2). The features they share is that each of them has a key space bearing the name of the partition from which data is archived (mainly ATLAS), the column names correspond to the time stamp of the IS information they are storing and the value is the value of the specific attribute stored on that row.

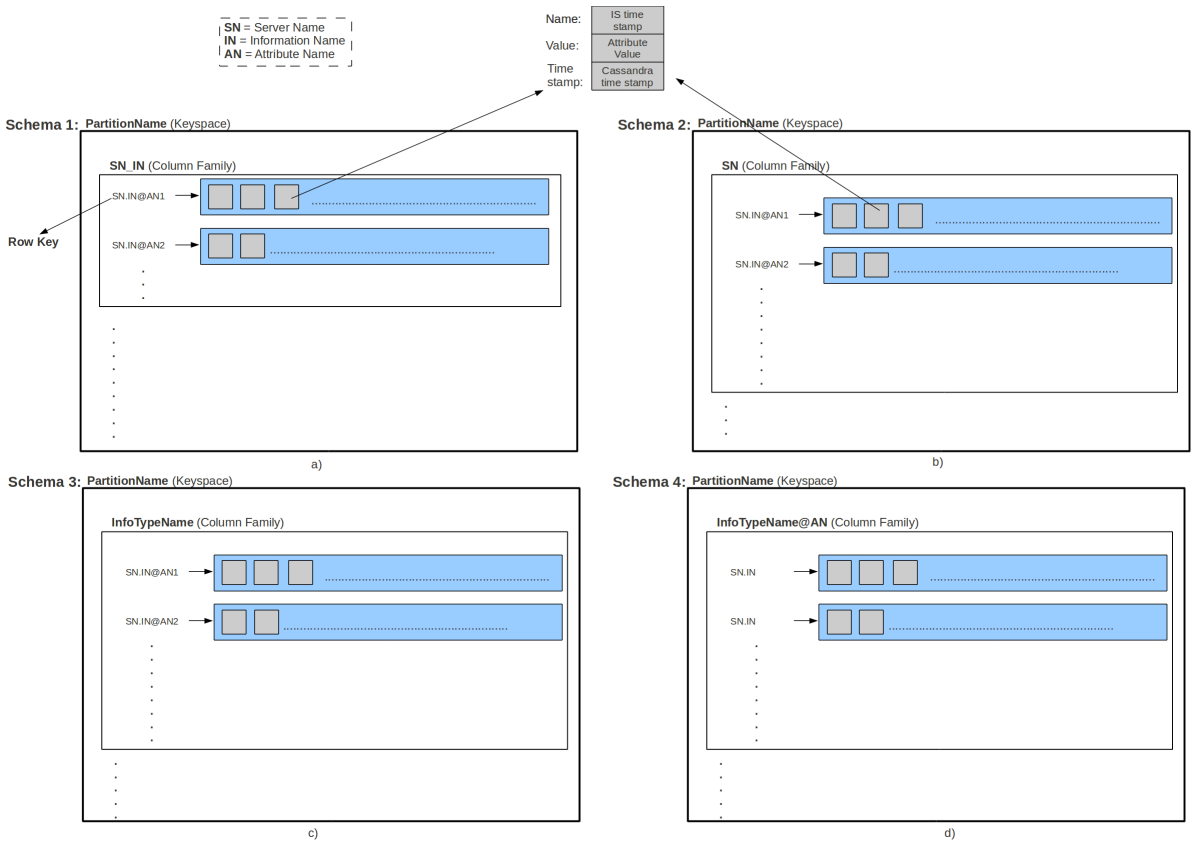


Figure 4.2: Cassandra Database Schema Alternatives

Schema 1 is the first one used and has one column family per IS information as already mentioned. The row keys are a composite of the full information name (which is itself a concatenation of the IS server name and the information source name - SN.IN) and the name of the attribute (the values of which are put in the columns of the row). For deployment in the ATLAS partition, it is estimated that approximately 170000 column families will have to be created. The query that is best supported by this arrangement is "Show the evolution of all attributes of an information source over time".

Schema 2 has one column family per IS server and the same row keys. Around 60 column families will be needed for deployment in the experiment. The query that suits this structure the most is "Show the evolution of all attributes of an information source over time".

For Schema 3 a column family has to be created for every information type giving an estimate of several hundred column families for ATLAS. The best handled query is "Show the evolution of all attributes of a type of information source over time".

Schema 4 has one column family for every attribute of every information type. This would give around 10000 column families while running in the ATLAS partition and would facilitate the query "Show the evolution of an attribute of a type of information source over time".

The table below summarizes the relative advantages and disadvantages of the schema layouts. The design decision that these demand is between accepting large latencies (evolution with IS is done by adding any new schema on fly) and modifying the schema structure so that far fewer column families are created which leads to more manageable schema creation times.

Schema	Advantages	Disadvantages
1	Better separation of reads and writes; Possibly better read performance (more CFs => less data per CF file); Less frequent schema rebuild needed	Very high schema creation time; Assures evolution by adding a totally new CF (time consuming op); Need to assume a large setup time to create most of the schema and then rely on the infoCreated() callback where to create new CFs (this will not scale if many infoCreated updates one after the other)
2	Low schema creation overhead; Assures evolution by adding a new col in a new row (which costs nothing)	While compacting the storage space used for one CF will double (in the worst case) – this can be significant compared to the total available storage space if the CFs are large; Need to create a new schema more frequently (split CFs) to separate data and keep read performance
3	Lower schema creation overhead than Schema 1.	Unbalanced CFs (some info types will have a lot of data while others will be small); Need to create a new schema more frequently (split CFs) to separate data and keep read performance
4	Possibly better read performance for the case when attributes of a certain information type need to be aggregated across all sources.	Very high schema creation time;

There is no silver bullet for choosing the schema (in view of providing the best retrieval performance), especially because of the way Cassandra is meant to be used: modeling data based on queries. The solution would actually have to be a mix of the schemas already described and possibly others corresponding to new or evolving use cases. In the circumstances of the intended use cases it is difficult to adopt a general schema that would provide a good fit for all retrieval queries possible. Moreover, whether a schema is a good fit or not for certain retrieval queries is something which needs to be quantified by doing performance tests. Without these the design cannot become any more accurate in predicting the best alternative to choose for the schema layout.

In consequence, the decision was made to mitigate the trade-off between manageable schema creation times (less structure in the schema) and the promise of faster retrieval queries (more structure

in the schema) in favor of the first option, since it is ultimately a matter of measurements to see what schema is good for what use cases (which are still developing anyway). Thus since prototype 3 a column family is created per IS server name. This significantly reduces the number of column families needed and satisfies the basic query mentioned in section 1.5.2 on page 12. It is a general structure to collect the data in. Depending on the retrieval performance it has and the completion of the use cases it is open to modifications or additions. This approach is not uncommon in industry. For example a company called Cloudkick which offers cloud monitoring services using Cassandra as a back-end explain in their white-paper [24] how a single column family for all monitoring data is used and from there that data is transferred to other column families modeled according to their retrieval use cases.

Also, the choice of a schema with lower build times was made towards satisfying the requirement that all the data from the specified IS servers is archived. If the application is started in one of the run control stages and it needs a long time to build it's schema, then by the time it finishes, it will have lost all the data that was published during that period. Moreover, each stage in the run control FSM has a timing constraint dictated by the dominating time it takes to configure or execute the lengthiest package. Figure 4.3 shows that the max time to execute the "configure" transition is about 250 sec and for the "start" is 100 sec. These times are a lot lower than the the time needed to build schema 1 for instance. Thus, having a low schema creation time gives a lot more flexibility of where in the transition phases of the run control to place the P BEAST.

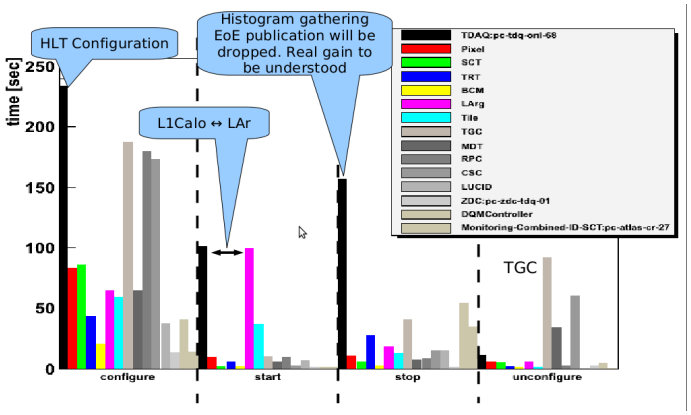


Figure 4.3: Run Control FSM Timings (courtesy of Giuseppe Avolio and Wainer Vandelli)

As for the meta data column families, their design started during prototype 2 with the introduction of the InfoSrcTypeByInfoSrc and AttrTypeByInfoSrcType column families. They provide the possibility of making queries without knowing the type of the data in advance. This is an important feature that in a sense makes the API self contained. It also satisfies a need that became evident only later on in the project. P BEAST needs to allow the user to stay on track with the evolution of IS data structures over time. For example an attribute with a certain name might have its type changed. P BEAST has to be capable of signaling this change.

Appendix 8.3 on page 53 shows the blueprint of the meta data column families in prototype 3. In addition to the ones mentioned in the previous paragraph, there is a Cache CF which aids P BEAST keep track of splitted rows when it is restarted. The RowKeyIndex uses the valueless column design approach of Cassandra to denormalize the information about all the keys in a certain column family, essentially providing an index to aid in retrieval.

4.2 Prototype design considerations

4.2.1 Prototype 0 - Basic gathering and insertion

This prototype represents a first attempt to experiment with the IS and Cassandra APIs and not much effort was put in its design stage. It was decided that all the code is placed in one class which embeds all the functionality from gathering information and inserting it in the database.

4.2.2 Prototype 1 - Optimization of information gathering

Apart from making the system more modular by separating the code that writes to the database from the rest of the code (adding DB interface), the design of this prototype is geared towards improving performance with respect to handling information input. The callback code has to be freed up as soon as possible so that the RequestProcessor threads in the Jacorb layer can take in new incoming requests. If time consuming operations are put in the callback code, for a burst of requests (which is to be expected from IS) the Jacorb thread pools would become empty very fast, threads being busy with executing the lengthy subscriber code. In this case, the processing of any new requests is stopped and the underlying buffers start getting full. If they do get full, requests are ignored and information published in the IS does not get archived in the database. Also, even though IS servers have buffers of their own, it is not desirable to put pressure on them and affect their operation. It is worth noting that the underlying CORBA communication between IS and the Jacorb client on P BEAST's side is implemented on top of TCP/IP. IS waits for a receive acknowledgment. After several unsuccessful tries the IS server will timeout for that specific information, it will place it in a queue and it will spawn a low priority thread that tries to resend information from the queue.

With all these considerations in mind and investigating the internals of Jacorb as well as research on server architectures and how multiple concurrent connections are handled, the executor design pattern is adopted. The basic idea is to use an in memory data structure to stack the incoming data and to permanently have several threads at disposal, which can execute some expensive operation (in this case the writing to the database) freeing up the buffer. This is graphically illustrated in 4.4.

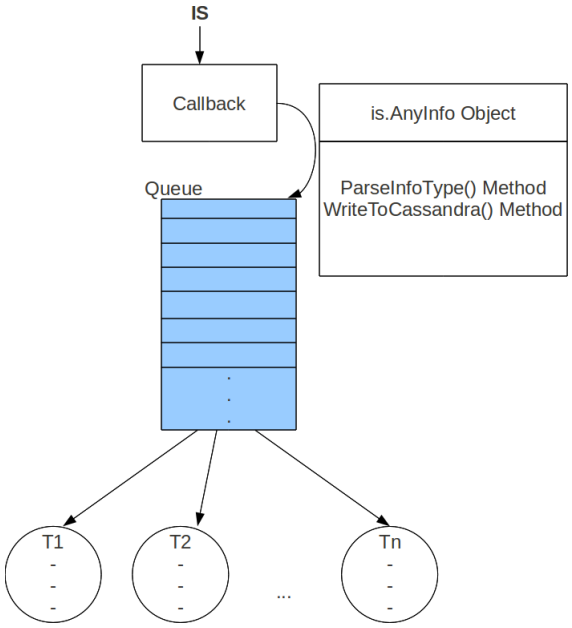


Figure 4.4: Executor design pattern

This approach fits perfectly with the top level functional blocks. The gathering component is the one filling the buffer with information received in the callback method (the AnyInfo object) and the processing component is responsible for emptying the buffer, its code (illustrated by the ParseInfo-Type() method in figure 4.4) being executed in the threads from the thread pool. Once the type of all the attributes in the information object received are determined and appropriate conversion to types suitable to fit in the database is made, the code of the insertion component (illustrated by the Write-ToCassandra() method) is executed in the same thread which blocks until confirmation is received from the database.

The two main mechanisms at play from which advantages are considered to be drawn are the intermediate buffering of incoming information and parallel execution of possibly expensive database insertion operations. The first of these will have the effect of lowering the load presented to the database at the expense of increasing the memory consumption in the periods of peak input traffic. Given enough memory and provided the database is able to accept input at a constant and sufficiently

high rate (in order to empty that buffer on average faster than the average rate at which the buffer is being filled) this mechanism will be able to absorb the load. Notice that in order for this to be absolutely guaranteed, the memory available and the database insertion rate have to cover the maximum possible average input rate. This latter constraint will have to be satisfied by appropriate choice of database platform and is taken for granted at the design stage. In case the first constraint is not possible to fulfill the buffering mechanism is still viable only with other extra techniques that have to be adopted on top. The load can be split across several application instances running in separate processes (on the same or on different machines). This will have the effect of allowing the buffer to be split in several sections which can grow across a potentially higher memory space such that peaks are absorbed. No extra design changes need to be adopted as this will be a purely implementation decision. The alternative is to design a delayed write mechanism on top of the executor pattern in which the information is written to a memory mapped file or in the local file system first (which is faster because of no network delays) and then it is read and sent to the database.

The second mechanism of the executor pattern is the pool of threads. They will execute insertions in parallel. The advantage of this is that while one thread blocks on the database call, another thread can start another transaction during the time in which the other one idles and performs no useful computation. In addition, having several threads lying around waiting for something to do (for information ready to be processed to appear in the queue) is faster than creating a thread each time it is required. Threads will be reused and will immediately start executing tasks without any delays in setting up.

4.2.3 Prototype 2 - Information processing and basic retrieval functionality

The additions brought by this prototype are in the area of filtering the information and retrieval queries, the last two basic functions necessary to have before moving on to do performance tests on a larger scale infrastructure. The experience with the IS API acquired in the previous two prototypes brought to light the fact that not all attributes of an IS information necessarily change value since the last update. Testing done on prototype two revealed many duplicate entries in the database. This caused an unnecessary increase in storage space which was fixed by designing a filter based on comparing incoming values with the last values inserted in the database for all attributes which are stored in an in-memory cache. On the retrieval side, a general query mechanism was designed around the idea of meta data (See discussion in section 4.1.2 on page 29).

4.2.4 Prototype 3 - Integration with partition infrastructure

The design of this prototype is the most elaborated. It adds on functionality to integrate with partition infrastructure, deal with unpredictable behavior of IS servers (polling mechanism) and adjust the length of column families rows to prepare for better retrieval queries. The subscriber code is made more modular in order to eventually ease deployment across several components if that is found to be needed later. Appendix 8.3 on page 54 shows the class diagram. The PBeast class contain the main application entry point. It instantiates a PBeastLoader object for every IS server of interest. Each of these objects instantiate their own ISInfoExtractor object which is the one implementing the IS callback methods and the ThreadPoolExecutor instance. In this way whenever an update callback comes, a new instance of the LoaderTask class is created and placed in the queue of the corresponding executor. Then it's run method is called by one of the threads. Each LoaderTask instantiates a CassandraDBClient object calling its insertion methods. It can be seen that the design allocates a thread pool for every server.

Chapter 5

Implementation

This chapter starts by describing the software development platform chosen. It then takes every prototype and presents details about how the design is put in practice, limitations and improvements compared to its predecessor. It gives pseudo-code examples to illustrate important sections or algorithms used, without going into very much detail. Implementation issues are presented and references to code are made where necessary.

5.1 Java Software Development Platform

5.1.1 General Considerations

Java is a well known programming language and development platform. It has been around for more than 15 years and there are many books, tutorials and on-line information about it. For this reason, this subsection only attempts a brief mention of Java as well as why and how the platform is used in the project, to make the analysis complete. More important is the discussion about Java multithreaded programs which lies at the heart of the implementation of P BEAST.

Java is known for its portability. This is achieved by the fact that any Java application or program is not directly run on a host machine but rather another process called Java Virtual Machine (JVM) is started being passed the necessary configuration parameters, the libraries used and the application that needs to be run. It is the JVM that communicates with the underlying OS passing and receiving signals etc.

The reasons for choosing Java as the environment for developing P BEAST are:

- the main database platform choice is also written in Java so it makes sense to stay in the same environment in case understanding of Cassandra internals becomes mandatory at some point or to avoid eventual inconsistencies between Cassandra and the API offered for other programming languages.
- previous experience in the language
- lots of built in functionality and support
- very good platform to use for writing multithreaded applications (see 5.1.2) which is needed to take advantage of the multicore machine where P BEAST will run and to improve resource utilization while waiting for database I/O

The IDE of choice is Eclipse. Apart form offering a lot of standard functionality like: code generation, IntelliSense etc which make programming faster and easier, it has its own compilers and can also be used to write applications for different target JREs.

5.1.2 Multithreading in Java

Java is a perfect choice to develop multithreaded applications because it has a memory model unlike C++ for instance, which does not offer one (yet). This has two major benefits: it allows programs that obey the Java memory model synchronization rules to function correctly on any platform (otherwise

concurrent program correctness is dependent on the processor used) masking low level hardware details form the application developer and it allows the JIT compiler to optimize multithreaded applications without fear of reordering code statements in such a way that would cause race conditions (these happen when the result of a task is dependent on the order in which threads execute the code of that task). Such a thing is possible due to synchronization barriers defined in the memory model. Basically to develop a multithreaded program the threads need to communicate and the way to do that is via shared memory variables. An order has to be imposed for the way those variables are accessed by threads. That order is intrinsically specified by the JMM.

When developing multithreaded applications in Java, the programmer needs to pay careful attention to the constructs used. It is important for correctness that the code produced be thread safe. There are a few general guidelines that can be used to make sure of this. The simplest is not to share an object's state amongst threads. If this is however needed, appropriate locking has to be employed. Abusing locking often introduces performance penalties, since the idea of using threads is executing as many things in parallel. Locking on an object can cause a chain of threads to wait on each other which in effect serializes program execution (as if only one thread was used). A sign of this would be that the CPU cores are idling. The program is considered successful from a concurrent point of view if maximum CPU utilization is reached (the application is CPU bound).

Java allows the programmer to explicitly invoke locking (by the use of the "synchronized" keyword) or implicitly by using special constructs that take care of locking behind the scenes. An example of one of these is the ThreadPoolExecutor class which implements the Executor pattern. This uses an input queue that synchronizes access from concurrent threads which push task objects. The tasks are automatically executed by available threads in the associated thread pool, effectively decoupling submission from execution.

5.2 Prototype 0

Prototype 0 consists of a simple exercise to subscribe to several hard-coded information sources in IS and store attribute values in Cassandra as they come along. It has a view, represented by the ISInfoProcessorPanel class, which displays the name of the information and updates of the number of values stored, giving a simple notice that data is flowing in the database. The class that does the work is called ISInfoProcessor. It assumes that a key space is already in place and creates the schema (one column family per information source) by subscribing to predefined information sources and doing an initial read of the values already there. Then, in the infoUpdated() method inherited from the EventListener interface of the "is" package, it determines the name and type of the information passed in and inserts the attribute values in the appropriate column family.

An interface is required for objects of a certain programming language that wants to perform remote method invocations using CORBA. The facility that allows Java classes to implement that interface is called Jacorb [26] and it is used by the developers of IS for the IS Java subscription API (which P BEAST uses). By importing the "is" package in the code of the IS client component (See appendix 8.9.1 on page 58) of P BEAST and using the subscribe method to register a callback handler (an object that implements the EventHandler interface) the Jacorb mechanism is set up behind the scenes. The way this works is that a POA object is allocated and is registered in the CORBA infrastructure to handle requests. In this case the requests are three types of events that come from the CORBA implementation on the IS side: create, update and delete. These are used to inform a subscriber that a distinct action occurred on a piece of information of interest. They are mapped to the InfoCreated(), InfoUpdated() and InfoDeleted() methods specified by the EventHandler interface. The code placed in these methods is the servant code that is executed by the RequestProcessor threads of Jacorb each time a new piece of information is created, updated or deleted. In short, the IS subscriber object defined in P BEAST is registered with the domain servers of CORBA and the three methods it implements from the EventHandler interface are in a sense executed remotely by the information server for which the subscription was made. See [27].

The limitations of this initial trial are obvious: single threaded, no modularization (code in one class), limited amount of information from IS is archived etc.

5.3 Prototype 1

As the design expanded into several classes (ISInfoProcessor, ParseAndWriteToDbTask, PbeastDBI, CassandraDBClient) to accommodate for the adoption of the Executor pattern and decouple the database, the implementation became slightly more complex. A java.util.concurrent ExecutorCompletionService object is instantiated in the constructor of ISInfoProcessor and is the Java construct used to implement the executor design. It is actually an improvement to the simple input buffer and thread pool approach because it adds an output buffer where results of executed tasks are placed. This implementation is chosen because it allows for post analysis of batches of results which might be needed in case certain tasks have to be re-executed. It is worth noting that the completion service internally uses an ExecutorService object and just adds the output buffer on top of that. Also, since the main purpose of using this mechanism is to cope with a bursty incoming workload, an unbounded input queue [31] is chosen.

The schema is still created by doing an initial read in the ISInfoProcessor class but the code in the update method is changed to make use of the CompletionService object:

```
infoUpdated(InfoEvent infoEvent) {
        extract the AnyInfo object from the InfoEvent object received
        wrap the AnyInfo object inside a ParseAndWriteToDbTask object
        submit the ParseAndWriteToDbTask object to the completion service
}
```

The AnyInfo object is the one holding the data received. As the title suggests it can belong to any information source to which a subscription was made. The ParseAndWriteToDbTask class implements the java.util.concurrent.Callable interface. Its call() method is executed by a thread in the completion service and contains boiler plate logic to determine the name and attributes of the AnyInfo object it contains, extract their value and pass them along to the appropriate method of the CassandraDBClient class which takes care of the insertion into the database. It is important to stress the fact that the callback method does not do any actual work, it just creates a task and places it in the input buffer of the CompletionService object.

The improvements brought by this prototype are: separates the code that writes to the database from the rest of the code (added PbeastDBI, CassandraDBClient), pool of threads now handle update callbacks using the same code as in the initial read stage (this code is part of the ParseAndWriteToDbTask which makes it more easily manageable and reusable).

The major limitation is that it was discovered that in the update callback there is no programmatic way of knowing which attributes belonging to the AnyInfo object actually change value (were updated) since the last time. So, attributes are always duplicated leading to a waste of disk storage.

5.4 Prototype 2

The main improvement brought by this prototype is using a cache (implemented by class AttributeCache) to filter out duplicates. In the call() method of the ParseAndWriteToDbTask class a comparison is first made with the cache entry for that specific attribute. Only in case the values differ the new one is placed in the cache and written to the database. The cache also maintains a counter for every attribute.

There were a few issues that arose when implementing this mechanism:

- the order of updates is not necessarily the one of the assigned IS time stamp (as mentioned in 1.1.3 on page 7). While this is not a problem from the storage point of view (because on the Cassandra side values are ordered based on time stamp anyway), cache corruption can be caused in certain special cases. For example when two updates are received very close to each other in time. It was however discovered that these cases are so rare that the programming effort to deal with them is not justified. The worst they could bring is duplicates.

- cache corruption can also occur because of concurrent thread accesses of the cache. This problem has been dealt with by employing locking. It was noticed that the issue only occurs for updates coming from the same information source very close in time such that they get processed in parallel threads at the same time. Since this is a rare event (as discussed previously) it was decided to employ locking on the information name which is the key of the cache, without fear of

performance penalty due to holding a lock for a potentially expensive write across the network to the Cassandra server.

The solution to the problem of duplicates is given by the call() method as follows:

```
call() {
 1) Get previous value from the cache for all the attributes of the information source.
 2) Obtain the lock for the cache entry
 3) Compare with new values and insert in DB only if the attribute changed value
in the update that is being handled by this task
 4) Insert the new attribute in the DB
 5) Replace the old attribute value in the cache with the new one
}
```

Prototype 2 is also the first prototype to have complete retrieval functionality, in accordance to the requirement that emerged relating to the user not knowing the type of the attributes queried for in advance. This is implemented by the method read() in class CassandraDBClient:

```
read(String SN.IN, String AN, Long startTime/endTime, Vector<PBeastColumn> result) {
1) Use SN.IN to get the type of the information source from the CF InfoSrcTypeByInfoSrc
2) Use the type of the information source and AN to get the type of the attribute from
the CF AttrTypeByInfoSrcType
3) Obtain the valid CF name (SN_IN)
4) Use SN.IN and AN to get all columns with time stamp between startTime and endTime
from CF SN_IN using the correct serializers based on the type of the attribute
5) Return  the type of the attribute so that the caller of the method knows what
Object values are passed back in the Vector result
}
*Notation:
        SN.IN = name of information source
        AN = name of attribute
        CF = column family
```

Even though this prototype covers all the basic functionality that P BEAST has to provide from an insertion and retrieval point of view, it has several limitations: it still relies on the initial read to create the schema, the configuration is hard coded, there is no actual integration with the partition infrastructure, all the callbacks are handled within one instance (which makes the code inflexible for deployment).

5.5 Prototype 3

Being by far the most complete implementation, prototype 3 brings many new changes and additions: a much more modular structure, several new mechanisms to deal with long term behavior, improved schema and more suggestive class names (ParseAndWriteToDbTask became LoaderTask; ISInfoProcessor became ISInfoExtractor), code to gather performance statistics.

Another important modification is the fact that the ISInfoExtractor class uses a simple ExecutorService to execute tasks. This was preferred to the CompletionService because the extra feature (the output queue) was found not to be of use. This change was minimal due to the fact that the CompletionService object was being passed an ExecutorService object anyway. As a consequence the LoaderTask5 class has to implement the Runnable interface instead of Callable and implicitly the name of the call() method changes to run().

The most interesting mechanisms employed are held in the implementation of two classes. Details are given in pseudo code as follows. The first important mechanism is in the PBeast class and it takes care to perform an initial subscription and then to poll the IS servers to see if the subscription is still valid:

```
main() {
        1) Read the configuration file
        2) Find the list of IS server names to subscribe to
                - filter the full list of IS servers defined in the OKS database
                against the regular expressions given in the configuration file
        3) Create the metadata schema and the data schema in the database
        (if it wasn't already in place)
        4) For every IS server of interest create a new instance of the
        PBeastLoader class calling its start method (to signal start of subscription)
        5) Sit in an infinite loop periodically forcing each PBeastLoader instance to
```

```
        check whether the server it is subscribed to is still running or not (by calling
        the checkAndRestart method of the PBeastLoader instance)
}
```

The run() method of class LoaderTask holds the implementation of the duplicates filter and the row splitting mechanism as well as maintaining the meta data column families necessary to accommodate the change in schema discussed in the design section 4.1.2 on page 29.

```
run() {
        Lock the AttributeCache entry corresponding the AnyInfo that is currently handled
        for each attribute attr of the AnyInfo obj {
                oldAttrVal = cached value of the attribute
                rowDelim = current row delimiter for that attribute (also stored in the cache)
                count = current number of values stored in the Cassandra row for that attribute
                        (also stored in the cache)
                rowKey = SN.IN@AN
                switch (attr->type) {
                        .
                        .
                        case (x): {
                                if(attr->value == oldAttrVal) {
                                        if(count == MAX_ROW_COUNT) {
                                                add new delimiter to rowKey
                                                reset count in cache
                                                rowDelim = new delimiter
                                        }
                                        rowKey = rowKey +  rowDelim
                                        add attr->value to batch insert with
                                                        key rowKey
                                                        serializer for type x
                                        oldAttrVal = attr->value
                                }
                        }
                        .
                        .
                }
                add oldCount and rowDelim to batch insert in Cache CF
        }
        execute the batch insert for the attribute values
        execute the batch insert for the Cache CF
        if any rows were split {
                insert them in the RowKeyIndex CF
        }
        insert the type of the AnyInfo object in the InfoSrcTypeByInfoSrc CF
}
```

In order to deal with all the possible IS types a case statement lies at the heart of the algorithm described above. IS published C++ types: boolean, s8/u8 (signed/unsigned char), float, double, s16/u16 (signed/unsigned short), s32/u32 (signed/unsigned integer), s64/u64 (signed/unsigned long) and string. They first of all need to be converted to Java types. Because in Java the concept of unsigned does not exist, the unsigned types are promoted to the next basic type (represented on more bits) with the appropriate sign masking. After that, the converted value is compared with the value stored in cache for that attribute name. In case of equality no further action is taken (the new value is ignored), otherwise the attribute value is added to the database batch insert and it replaces the old cached value. A decision is also made whether the maximum number of values were already stored in the Cassandra row corresponding to that attribute. If this is true then the row is split up by resetting the count associated with that attribute and adding a delimiter which is equal to the time stamp of the last value added in the database (for that attribute). This is the minimum amount of information needed to fully reconstruct a temporal retrieval query across rows that are split (See algorithm of method readAcrossSplittedRows of class CassandraDBClient). Only after looping across all attributes, the batch insert of attribute values is executed. Also, the meta data column families are updated (if necessary).

These meta data column families are mostly used by the new retrieval function except for Cache which is used at application start-up to rebuild the count and row delimiter parts of the in memory cache for each attribute. This allows the data to be inserted in the rows with the appropriate delimiter.

The readAcrossSplittedRows method of CassandraDBClient improves the read() method of prototype 2:

```
readAcrossSplittedRows(String SN.IN, String AN, Long startTime/endTime,
Vector<PBeastColumn> result) {
        1) Use SN.IN to get the type of the information source from the CF
        InfoSrcTypeByInfoSrc
        2) Use the type of the information source and AN to get the type of
        the attribute from the CF AttrTypeByInfoSrcType
        3) Obtain the valid CF name (SN)
        4) Identify the keys that have:
        - the max timestamp (delimitor of the splitted row) < startTime
        - the max timestamp (delimitor of the splitted row) < endTime
        5) Perform a Cassandra multislice query to retrieve all values from
        these rows and place them in result
        6) Return  the type of the attribute
}
```

Chapter 6

Testing and Deployment

6.1 Overall testing procedure

Testing is separated in several stages. This is facilitated by the different levels of partition simulation that the ATLAS Online system can provide through its general and extremely flexible configuration and inter process communication mechanisms. Partitions can be created for different purposes and for various infrastructure scales. In the case of this project these scales resume to the following:

Local machine partition: a few tens of applications all running on the development workstation

Preseries partition: a few hundred applications spread across a cluster of machines put at the disposal of ATLAS TDAQ developers for testing purposes. This is called the "Preseries" cluster and is meant to be a smaller replica of the full computer infrastructure of the ATLAS experiment

ATLAS partition: ~30000 applications running on the experiment infrastructure in the enclosed network (Point 1). Note that the ATLAS TDAQ system has not reached its final size yet.

The focus of the tests depends on the scale of the partition where they are performed. The local machine partition serves as a platform for functionality checks and routine bug fixing, with performance and integration tests being reserved for the larger scale partitions. This distribution is primarily governed by the amount of data each partition is capable of generating and publishing in its own information service. Since the local machine partition is small it makes sense to only attempt performance tests at higher scales where the system can really be pushed to its limits by the data flow. It also came by necessity because, for instance, it is more time consuming to make tests on a partition simulating the entire data flow on the preseries cluster of machines. On one hand all the files necessary to run the application (configuration files, jar files, test result files etc) need to be transferred back and forth between the working machine and at least one of the cluster machines via SSH. To complicate things in order to make sure that everything works, multiple shells need to be opened to have access to the remote machines running P BEAST and the ones running the Cassandra server. This would involve opening a lot of ssh sessions from the workstation and a lot of password writing. Luckily there is a tool called Byobu which makes handling the multitude of shells a lot easier to manage (See 6.2 on the next page). Nevertheless, the testing procedure is more involving, not to mention the the fact that the preseries cluster may not always be available due to other tests being performed there. Planning of tests needs to be done in advance and the machines to be used need to be booked on an internal page.

Tests destined for the experiment infrastructure are a completely different story. They are usually done within periods of technical stop. These happen approximately every two months and last for about a week. During these periods the LHC does not offer stable beams and the ATLAS detector is not taking data but rather checks, tests and calibrations are done on different subsystems. However, due to the nature of the project (that is to archive data published during data taking sessions of the ATLAS experiment), P BEAST tests are not bound to technical stops. It is preferable that they are performed during actual operation of the detector. To be able to do this, special access needs to be granted inside the ATLAS enclosed network and machines where to install and run P BEAST and Cassandra need to be allocated. Security within the ATLAS enclosed network is taken very seriously and each time a user logs in remotely to make some tests, even if he owns all the necessary permissions,

verbal acceptance needs to be given by the shift leader that is supervising the data taking session and the control of the detector at that time.

It is also worth mentioning that throughout the development a lot of component testing was done such as Cassandra benchmarking or stress tests using a ready available IS information generator. The latter ones are mentioned in section 6.5.2 on page 49 together with some results.

The testing procedure is as follows: after being designed and coded, every new prototype is integrated with the local machine partition (valid for the later prototypes because the earlier ones were started manually) and then its functionality is tested. After the necessary fixes are made and satisfactory results are obtained (visual checks systematically recorded in tables) a move is made to test performance on the preseries partition. This cycle (of tests on local and then preseries partition) continues until a prototype with a complete functionality, with respect to the requirements and the feedback accumulated from previous iterations, is shown to handle the information flow in the preseries partition by performance measures. When this objective is met, the system is ready to be tested on the ATLAS partition.

The set of functions that needs to be reached in order to make the transition to the full experiment is:

- handling IS dynamic behavior
- adaptive storage and retrieval mechanisms

6.2 Testing tools

Several tools were used to aid in testing P BEAST. Some of them are custom made and are described were necessary in the testing sections that follow. The majority however is composed of widely available tools for debugging, monitoring and performance evaluations of applications and make the object of this section.

6.2.1 Cassandra utilities

These are probably the most specialized of the widely available tools used to check the functionality of P BEAST.

"cassandra-cli"

It is a simple command line interface client that ships with the Cassandra server. It can be started by invoking the command: **"*cassandra-cli -host localhost -port 9160*"** while in the /bin directory of the Cassandra root installation folder. It takes as arguments the name of the machine where the Cassandra server is running (in this example that is the same as the machine where the client is running) and the port on which it is listening (default 9160). It provides a set of simple commands to directly interact with the database, allowing the user to view available key spaces ***"describe keyspaces;"***, log into a key space **"use 'Keyspace Name';"**, create or delete column families, insert columns in column families or perform retrieval queries ***"get 'Column Family Name'['Row key'];"*** the results of which are shown as text in the shell where the client is running. This client is extensively used to perform visual checks of the data stored in the database.

"nodetool"

This is a program that is useful when configuring a multi node Cassandra cluster and was used extensively during set up of the three node cluster benchmarks. Some examples of how this utility was used are:

- **"*nodetool decommission -h pc-preseries-xpu-004*"** pulls node with name 'pc-preseries-xpu-004' out of operation after copying its data to another node in the cluster
- **"*nodetool -h pc-preseries-xpu-003 ring*"** displays information about all the nodes in the cluster to which 'pc-preseries-xpu-003' belongs to (IP address, amount of data stored by each node in GBytes, percentage of tokens a node owns out of the total token space, the range of tokens assigned to this node). The output looks like this:

```
Address Status State Load Owns Token
10.149.38.21 Up Normal 4.81 GB 47.39% 50733617181936173...
10.149.38.23 Up Normal 2.94 GB 27.74% 97929319344660722...
10.149.38.20 Up Normal 10.4 GB 24.87% 14024094314821396...
```

But it can also offer statistics about server operation by passing in options like tpstats (thread pool statistics) or cfstats (column family statistics). The latter was used more frequently to check and understand server operation. The output given by invoking the command "***nodetool -h pc-preseries-xpu-004 cfstats***" during the operation of the server can be seen in appendix 8.5 on page 54. It gives lots of useful statistics like internal server operation like the mem-table switch count which indicates how many memtables have been flushed to disk.

6.2.2 Byobu [32]

Byobu is an open source program that extends the Screen window manager. It provides a simple to use interface for having multiple bash shell tabs open across the same SSH connection. These appear inside the same shell from where the session is open. It made performance testing on remote machines considerably faster and more efficient. It also has a suspend feature which allows a Byobu session to stay on the remote machine if the SSH session is closed.

6.2.3 "log4j" [33]

Log4j is a logging facility developed by Apache. It provides different levels of log messages and an easy way to set the range of levels to be written to the log files. This is extremely useful because it means that if for instance an application which usually writes INFO messages to its log file all of a sudden needs to be debugged, it is sufficient to change the threshold level in a configuration file to DEBUG without any intervention to the source code. This will cause a wider range of messages to be put in the log file giving a better idea of what the application is doing. The mechanism that allows this is simple: a Log object is instantiated with a certain name (in order to be unique this is usually the full class name), messages are written in the code using the methods of this object instead of System.out.print(), the name of the method used describes the severity level of the message (trace, debug, info, warn, error, fatal), in the configuration file a logger entry is defined for the name defined in the code and is linked with a certain appender entry which defines where the message is printed (a custom log file, stdout, stderr).

6.2.4 "perf4j" [34]

Perf4j is a library that can be linked with log4j to record measurements of Java code. The timer object records statistics for a block of code between when its start method is called and its stop method (or lap). For example, the code below can be used to record statistics about the execution of method 'method' from class 'Class1':

```
public Class1 {
        void method() {
        LoggingStopWatch stopWatch = new Log4JStopWatch(
                Logger.getLogger(Class1.class.getCanonicalName() +
                                                        ".TimingLogger")
        );
        stopWatch.start();
        //...other code
        stopWatch.end();
        }
}
```

By creating a logger entry with the name "Class1.TimingLogger" in the log4j configuration file and liking it with a special king of perf4j appender, the transactions per second (how many times method() was called per second), mean, max or min execution time of method() can be recorded every predefined

period of time (time slice). Nothing is recorded if the function is not actually called at all within a time slice. Because of this in order to produce correct plots extra manipulation of the .csv files was required in order to fill with zeros values of the statistics at points where the function was not called.

6.2.5 VisualVM [35]

VisualVm is a monitoring tool for Java applications. It attaches to a selected instance of the Java Virtual Machine and displays real time information about different aspects of interest like heap memory consumption, garbage collector activity, CPU usage, number of threads, number of classes etc. It can also take snapshots of the memory or threads used by an application. Examples of utilization can be found in the section 6.4.

6.3 Local machine testing

There are many information types that are published in the IS servers of even a small partition. Each has several attributes of different types. Testing whether every single attribute coming into the system is archived correctly is a challenge. To deal with it, custom programs were written:

CassandraReader - a command line utility that prints to the screen all the values currently stored in the database for a certain attribute, the number of results returned, the type of the values and the time it took to execute the query.

ConsistencyChecker - a program that first reads the attribute counts from a file dump of the in memory cache (performed at the end of a data taking session). Then it uses the read method of the CassandraDBClient to return all the actual values stored in the database for all attributes and keeps track of the number of results returned. The two lists are compared to see if the numbers are equal. This constitutes a crude way of checking that the cache and storage mechanism works.

These tools were used in two main testing approaches:

- observations recorded systematically

  A range of attributes covering several IS servers and information names as well as all basic types is selected. These are chosen to cover both fast varying and slow varying values. The CassandraReader and Cassandra-cli are used for all of them to perform visual inspections of the data that is actually stored in the database. Results are recorded in tables.

  This method of testing was used in building check lists for prototype 2, to see whether all attribute types are archived and can be retrieved correctly according to the mapping between C++ and Java types. It was also used for verifying correct operation of the row splitting mechanism employed in prototype 3.

- automated consistency check

  This implies running the consistency checker at the end of a data taking session. It was used in the testing of prototype 2 on the preseries cluster, to verify that duplicates are not stored and that the in memory attribute cache is consistent with what is stored in the database.

6.4 Preseries cluster testing

6.4.1 Procedure

The log4j logs are a good way to see that P BEAST is actually running. Messages are printed to the logs at key points along the code to indicate that the application successfully accomplished a certain task or reached a desired stage. To get more in depth information about the performance of PBEAST while it is running it is necessary to monitor the Java virtual machine memory and CPU usage. This is done with a specialized monitoring tool called VisualVM which attaches to the JVM and displays customizable views of various information about the applications running. For the case of P BEAST the most useful facilities are the memory panel showing a real time graph of the used heap and the total available heap, the threads real time graph (See 6.1 on the next page) and for seeing what happens

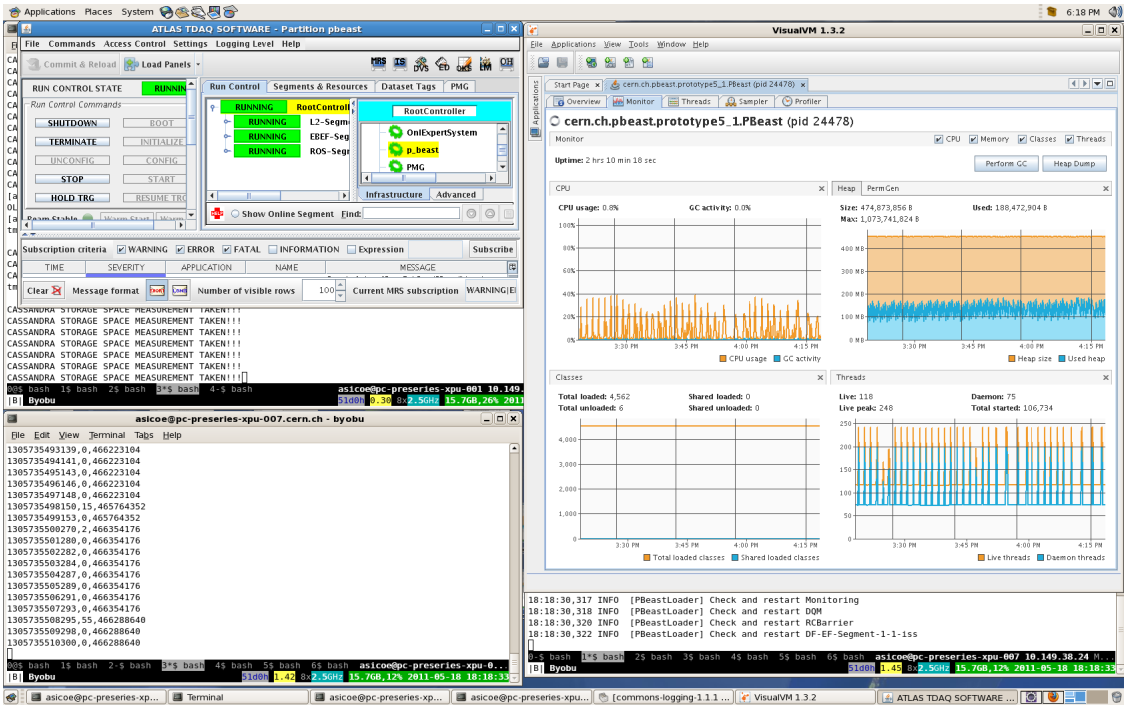


Figure 6.1: Snapshot of an on going preseries test

under the hood of the APIs used, the time line of the running threads (See appendix 8.4 on page 55 for a view of the Jacorb threads running servant code from P BEAST).

A snapshot of how a typical workspace looks for a preseries test is given in 6.1. Appendix 8.5 on page 55 shows the threads of the thread pools used by the ThreadPoolExecutor objects.

Until now the means used to observe the real time behavior of the application have been described. However evidence is required to show the behavior of P BEAST and how it relates to that of IS. Two kinds of tools were used to gather measurements during operation and store them in files: perf4j/log4j and custom written programs.

The way perf4j works in conjunction with log4j to acquire statistics about a block of code is described in section 6.2.4 on page 42. This mechanism was used to record the following:

- mean execution time of methods createDBDataSchema() and createDBMetadataSchema() of class PBeast. This is useful to keep track of schema creation times which is one of the reasons why integration with the partition infrastructure is problematic (See discussion in 4.1.2 on page 29).

- frequency of execution and mean execution time of methods infoUpdated(), infoCreated() and infoDeleted() of class ISInfoExtractor (per IS server). These give measurements of IS data rates and are important in understanding behavior and offered load for individual servers of interest or the sum across all of them.

- frequency of execution and mean execution time of successful execution of the run() method of the LoaderTask class (per IS server). These measurements reflect the successful insertions done to the database and constitute an important indicator that indeed the database is able to take in the offered load.

- frequency of unsuccessful execution of the run() method of the LoaderTask class (per IS server). These measurements reflect the insertions which were not successfully because the database had trouble coping, is down or communication with the database is down. If everything goes well it should always be zero.

For other more specialized measurements custom programs were written. These constitute of a simple thread that once started, periodically wakes up, takes the necessary measurement, writes the new value in a .csv file and then sleeps for a predefined time. The ISInfoExtractorStatsGathererThread

class is such an example. It is started in the main method of the PBeast class after all the Loader objects have been initialized for every IS server. It samples the length of the input queues of the ThreadPoolExecutor of every ISInfoExtractor object and writes individual values to a file as well as the sum of all the lengths and the total memory used by the JVM at that time. These measurements are useful for keeping track of memory usage and discovering eventual memory leaks, as well as indicating whether the input queues are freed up in a timely fashion.

Another example is the and CassandraStorageSpaceStatsGathererThread class. This is a standalone application and has to be invoked on the machine where the Cassandra server is running. It starts up a bash shell process using the RunTime Java class and executes the Linux "du" command used for checking the size of files and folders on disk. It then reads the lines returned from the output stream of the bash process and parses them to extract the sizes of the folders of interest which it writes to a predefined file together with the time stamp at which the measurement was taken.

6.4.2 Results

The performance tests gathered the same statistics but varied in duration. Shorter tests (approximately 40 min) were done for reasons of plotting comprehensible graphs with manageable number of data points spanning a full partition cycle. These are given below. Longer tests of three to four hours were meant to check memory consumption is in control and the system exhibits constant behavior over time. They also provided enough data to visualize the increase in storage space during a lengthier data taking session.

Another important thing to mention is that the performance graphs are all displaying measurements (punctual, averaged over a certain interval) against the time when they were taken. Due to the fact that different tools were used to get graphs, some of them will have human readable time stamps as labels of the x axis (these are the graphs obtained with perf4j) or absolute timestamps (the graphs obtained with the custom made tools).

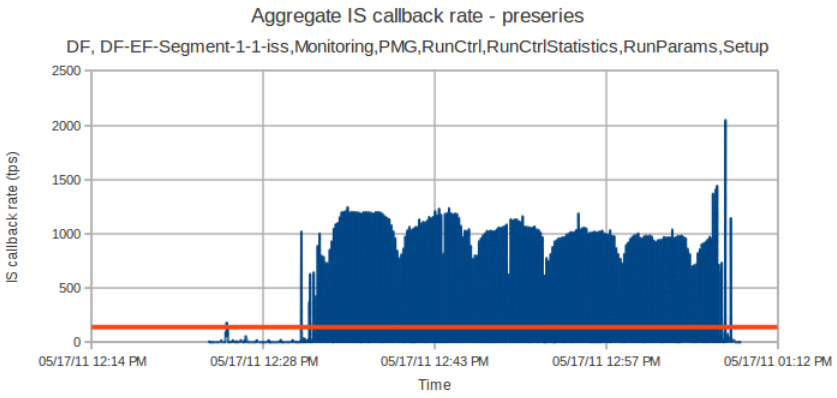


Figure 6.2: Aggregate Update Callback Rate - averaged every 1 sec interval

There are two graphs presenting the measured IS update callback rates (transactions per second). The difference between them is the counting interval. Graphic 6.2 shows the total number of update callbacks from all IS servers from which data was archived (the server names are: DF, DF-EF-Segment-1-1-iss, Monitoring, PMG, RunCtrl, RunCtrlStatistics, RunParams, Setup). A measurement is taken every second. This is achieved by setting the 'TimeSlice' parameter of the perf4j 'CoalescingStatistics' appender, specified in the log4j configuration file, to one thousand. It can be seen that the rates vary a lot. This one second interval average is perfect for capturing the immediate behavior of IS. The same measurement only with an averaging interval of ten seconds (TimeSlice of 10000) is shown in figure 6.3 on the following page. It shows the broader behavior of IS with callback rates which stick around the mean most of the time with larger variations during initial and final transitions of the partition (beginning and end of the graphic). This happens because during these phases update callback rates increase especially for the infrastructure IS servers like RunCtrl, PMG, DF that need to keep track of all the sudden additions or deletions of information from various applications that either start appearing or disappearing. As expected the mean line of the two graphs is the same (note that

different data taking sessions were used to record each of the graphs).

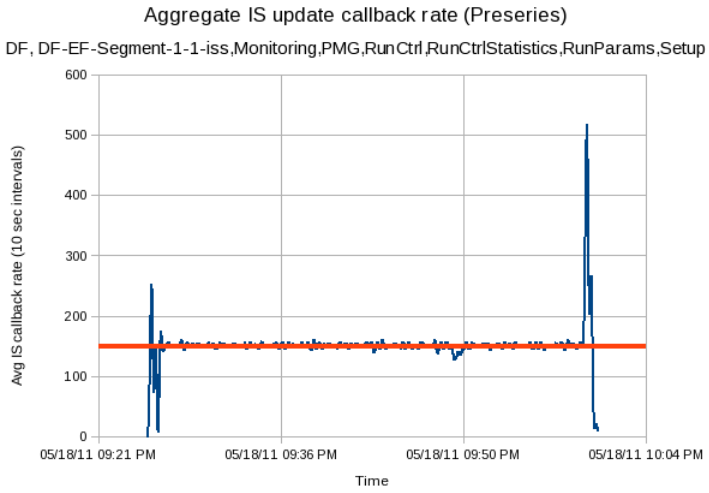


Figure 6.3: Aggregate Update Callback Rate - averaged every 10 sec interval

Another interesting aspect is the average execution latency of the run() method of the LoaderTasks being executed (See figure 6.4). They are evaluated at ten second intervals. It is important to have these statistics per IS server, to be able to distinguish between them. This becomes important in case the application needs to be split in several instances when deployed in the experiment partition. It would allow reasonable grouping of IS servers per instance. Also, by comparing these measures on the preseries partition with the same measures on the ATLAS partition, useful indication can be obtained whether the Cassandra server can actually absorb the writes. If that is true it is expected to see that the latencies do not vary at all or by only a small amount when moving from preseries to experiment.

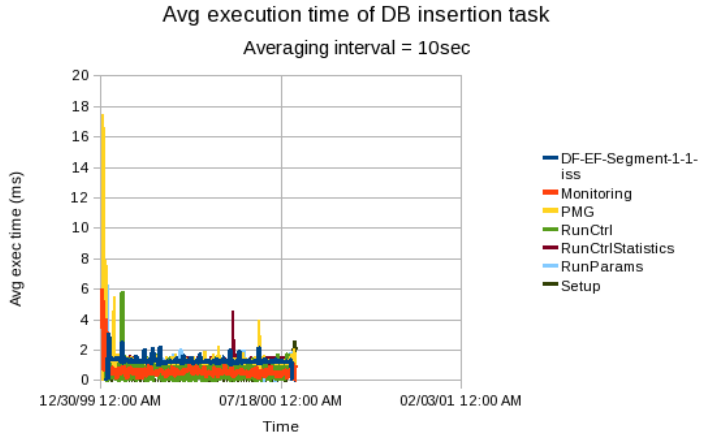


Figure 6.4: Loader Task Execution Time - averaged every 10 sec interval

The previously discussed test results are useful to quantify and understand behavior of incoming data and processing latencies. By comparing with similar measurements done on the ATLAS experiment partition, valuable approximations of the difference in scale can be obtained which allow for optimal structuring of the application for deployment. The 3400 ops/sec mean rate of the ATLAS IS (See fig 1.3 on page 8) is approximately 25 times higher then the mean rate of the preseries partition IS severs (150 ops/sec). Keeping in mind that P BEAST is intended to be used to store information for around 50 IS servers in the ATLAS partition (see discussion in section 1.1.3 on page 6), a better estimate of the mean input information is scaling by a factor of two to get 6700 ops/sec. Putting this figure in the context of the Cassandra benchmarks shows that it is within the maximum load (10000 ops/sec) that can be handled by a one node cluster (See graph in fig 3.7 on page 26). This is an

indication that Cassandra running on a single machine can handle the information input but because the average is in the upper bound of the supported range, it means that the system will have to rely on buffering in P BEAST when peaks occur. A three node cluster however would not put so much stress on the buffers because the estimated average input rate is comfortably situated in the middle of the input range found through benchmarking (20000 ops/sec as can be seen in fig. 3.8 on page 27).

A direct indication of the fact that the system is absorbing the information is represented by the measurements of the execution queue sizes. Figure 6.5 a) shows total queue sizes measured every second. It can be noticed that the queues are being emptied from one second to the next and that the thread pools are disposing of the tasks inserting data in the database, in a timely manner. Figure 6.5 b) represents queue size measurements per IS server. It shows for instance that DF (Data Flow) is the most active server during operation. At the end of the data taking sessions the RunCtrl server is more active.

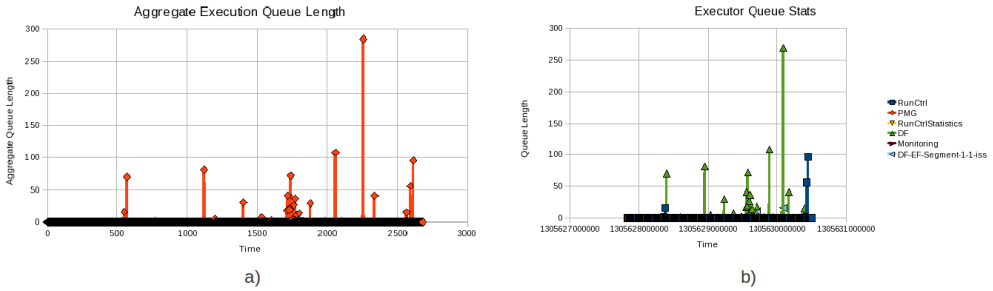


Figure 6.5: Measurements of Executor Input Queue Sizes: a) Sum for all IS servers b) Per IS server

The last performance measurement took is the storage space. This is measured at 20 minute intervals and by extrapolation an indication of the amount of storage space needed for long term usages of the system (per year) can be estimated. Figure 6.6 gives a 800MBytes increase in storage space every 3.6 hours which translates to a storage rate of 62 KBytes/sec. Multiplying this with the typical yearly run time ($8 * 10^6$sec - 100 days of experiment run time per year. See section 2.8.2 on page 17) gives a result of 0.496 TBytes/year stored by gathering data within a partition running on the preseries cluster. To get the estimate for the ATLAS experiment partition a scaling factor needs to be applied. To find this, the ratio between the real ATLAS partition estimated information rate (20MBytes/sec, 170 Tbytes/Year See section 2.8.2 on page 17) is divided by the preseries cluster the average rate of information coming in PBEAST from the the preseries partition (300 KBytes/sec). So, the scaling factor between ATLAS and preseries data rates is 66.67. Multiplying this with 0.496 TBytes/year gives 33 TBytes/year estimated storage space needed for when P BEAST operates in the production environment. It is significantly lower than the initial estimate of 170 TBytes/year. This is the direct consequence of the duplicate filtering applied.

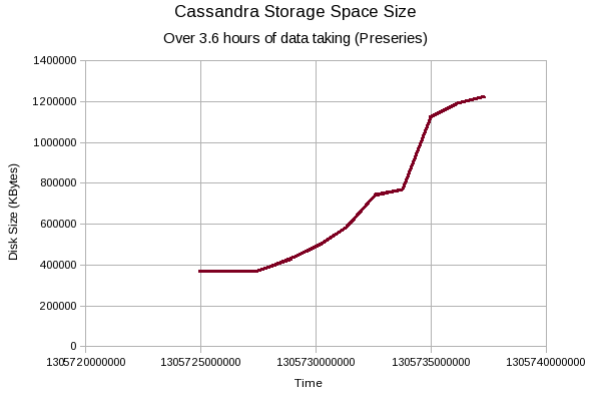


Figure 6.6: Cassandra Storage Size Measurements

6.5 Experiment testing

6.5.1 Procedure

The procedure and tools are the same as those used for the preseries cluster testing (see section 6.4 on page 43) with the addition of a simple utility (bash script) that periodically dumps to a file the load average of the machine where the Cassandra server is running. This is one of the measures given by the Linux 'top' command. It shows a one minute average of the work done by the computer.

The hardware setup consists of two quad core machines, each of them equipped with several RAID disk arrays. This detail is important because it allows optimization of the writing performance of the Cassandra server. The commit log and data directory are placed on different disks . This separates the write and read I/O operations, allowing writes to pour in without being disrupted by compaction and other operations performed on the data SSTables.

6.5.2 Results

During the first attempt of storing data at production rate, a single P BEAST instance was used to see how much load can be sustained in a realistic environment. It was possible to show that a P BEAST instance handles without trouble an average load of 2000 ops/sec. Figure 6.7 is a plot of the measured update callback rate across some 40 minutes of data taking. In comparison, the maximum load that can be sustained by one instance using the IS Java API in the exact same configuration as PBEAST (only without any servant code) is 20 000 ops/sec.

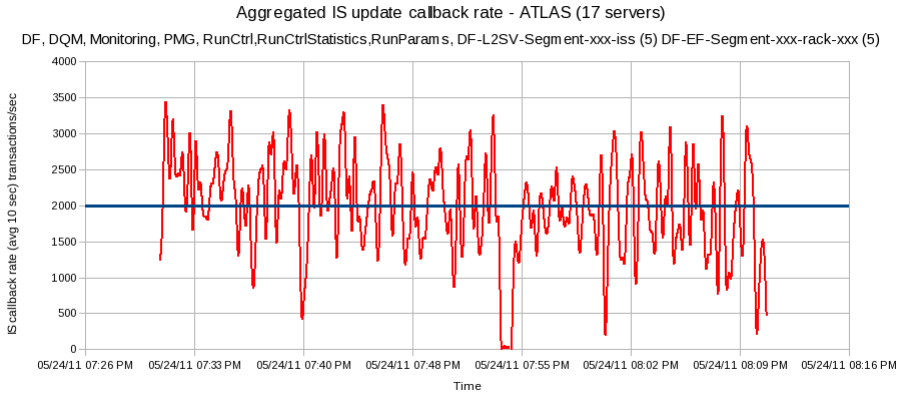


Figure 6.7: ATLAS Aggregated Update Callback rate

It was also shown that the load on the machine running the Cassandra server remained low during the data taking session. Figure 6.8 emphasizes a measured average load of below 0.8 out of maximum 4 which can be obtained on a quad core machine. Since this measure takes into consideration not only tasks that are ready to be run on the processor but also tasks in uninteruptible sleep (waiting for I/O), the low figure is an indication that the Cassandra server is capable of processing a far larger load.

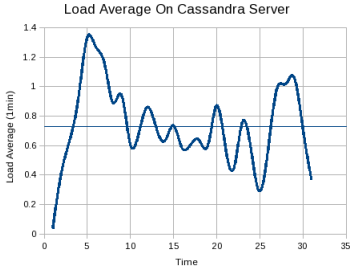


Figure 6.8: Load Average on Cassandra Machine

Due to the fact that the performance of the P BEAST instance was slightly lower than expected, it was decided to better investigate its behavior instead of immediately scaling up the system with sufficient instances to cope with the IS rate coming from ATLAS. By tuning several parameters in the CORBA broker it can be shown during a dedicated test that the situation can be improved to the point that 5000 ops/sec can be sustained. Figure 6.9 shows the measured update callback rate every 10 second interval.

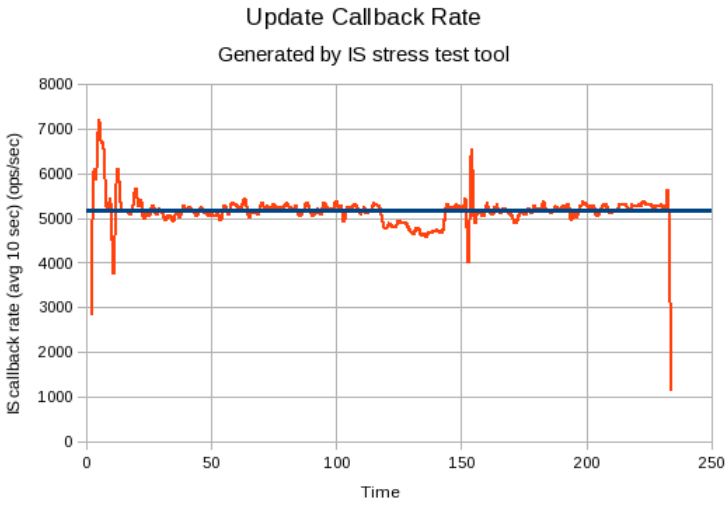


Figure 6.9: Dedicated Update Callback Rate

A final validation of the system with multiple P BEAST instances and possibly a networked deployment of Cassandra on multiple nodes within the ATLAS infrastructure is still pending, but there is sufficient confidence that all elements are present to be able to tune and optimize the different components and achieve the final goal of being able of archiving all IS information produces by the ATLAS experiment.

Chapter 7

Conclusions and Outlook

This report presents in detail the work done in order to provide a persistent back end to the operational monitoring information of the ATLAS TDAQ. P BEAST consists of five main functional blocks: data gathering, processing, insertion, storage and retrieval. A modular and scalable system was developed which is capable of satisfying all the main requirements that were gathered during the initial phase of the project. In particular, the choice of the Cassandra technology as the platform for data organization and storage proved to be very effective, thanks to its flexible schema, to its massive data insertion capabilities and the horizontally scalable storage system. Cassandra was studied and bench marked in detail and the use of such a novel technology constitutes one of the highlights of the work done for this project.

The design of P BEAST focused on modularity in order to be able to study all individual functional blocks independently and ensure the correct performance of each stage. The capability of scaling the data insertion part by instantiating multiple copies of it was planned for from the beginning. This is particularly important because it allows the system to account for the fact that the the TDAQ hardware and software infrastructure will continue growing in the next few years and thus the amount operational information data to store will increase. As LHC breaks record after record in beam luminosity, generating more events in the detector, pressure is put in the data handling infrastructure. More processing power will be soon added to the Event Filter system: 15 racks of 32 machines each. This will increase the amount of information published in IS and thus will put more load on P BEAST which thanks to Cassandra will have the means to scale accordingly with limited administration and maintenance work.

During the execution of this project several challenges had to be faced. The first one was integrating in a completely new working environment. This was facilitated by participation to weekly group meetings where occasionally P BEAST status presentations were held. The second consisted of requirements gathering which consumed a significant portion of the initial efforts and had as outcome a requirements document revised by the group. The third major challenge was the choice of storage platform and the last one was understanding the scale of the problem which became clearer as measurements and tests were done on infrastructures with increasingly larger scales.

The project involved a continuous dialog with the "clients" which are experts and developers working in the ATLAS TDAQ group, the same people that offered guidance and support, where necessary, in different technical areas. Keeping the project moving forward in a fast and interactive way was an objective from the start and the iterative approach adopted made possible to put this in practice. It allowed the project to develop rapidly and gave room for the dynamic interaction with the users. Evolution at an appropriate pace was largely due to this. The work flow had flexibility of expanding or contracting depending on the issues encountered.

The project is now in an optimization phase. More studies are needed to tune the deployment of P BEAST in the ATLAS experiment in terms of gathering instances and Cassandra nodes. Further work needs to be done in in adding configurable and pluggable filters. After that the focus will shift towards the data retrieval components (query performance, integration with data clients - dashboards, GUIs). Once these features are in place investigations will need to be done in organization of data for long term storage.

Chapter 8

Appendix

8.1 Target IS information

Server Name	Number of information objects published
DF-EF-Segment-...	3224
DF-L2-...	9341
DF	2768
DDC	3
DQM	107124
Monitoring	2233
RunCtrl	21686
RCBarrier	4
RunCtrlStatistics	22
Setup	6655
RunParams	121
PMG	12424
Resources	5100
DDC	24
TOTAL =170729	

Table 8.1: List of ATLAS IS servers holding operational information

8.2 IS Server Rates

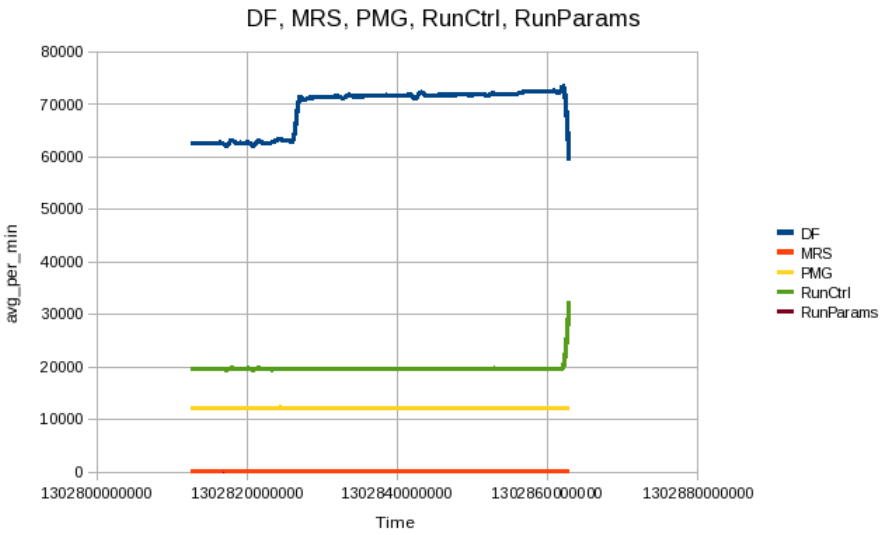


Figure 8.1: Infrastructure IS servers counts per min

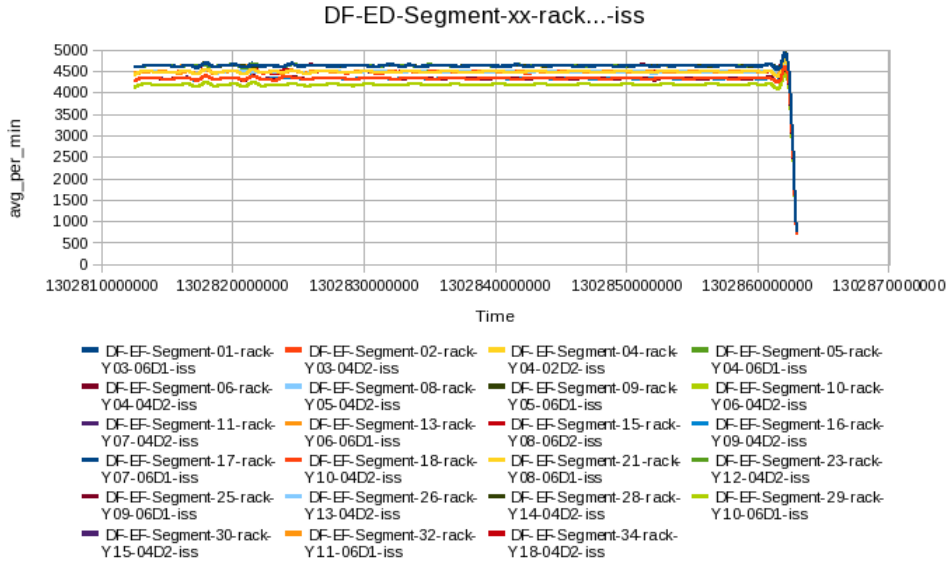
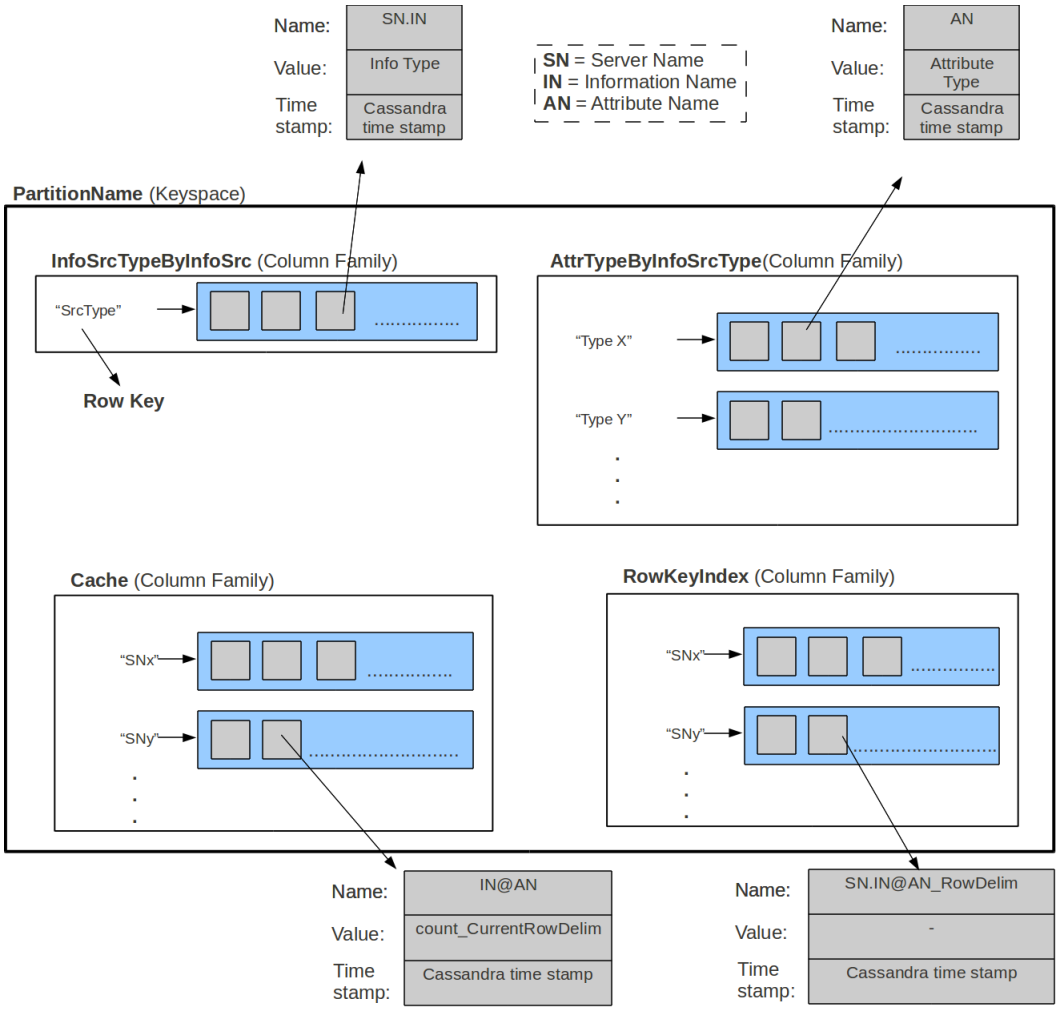


Figure 8.2: DF-EF servers counts per min

8.3 Medatada Schema Prototype 3



8.4 Prototype 3 Class Diagram

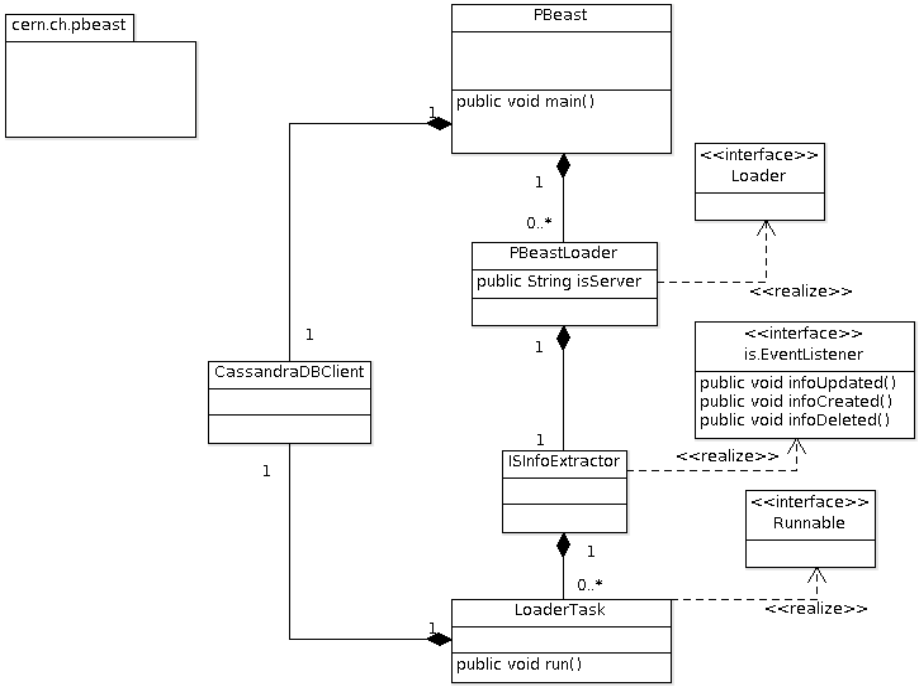


Figure 8.3: Class Diagram Prototype 3

8.5 Cassandra "nodetool" example output

```
Keyspace: usertable
Read Count: 873783
Read Latency: 0.43422425934127806 ms.
Write Count: 3490007
Write Latency: 0.03613370116449623 ms.
Pending Tasks: 0
Column Family: data2
SSTable count: 5
Space used (live): 5166834128
Space used (total): 5166834128
Memtable Columns Count: 1052066
Memtable Data Size: 127299986
Memtable Switch Count: 2
Read Count: 873783
Read Latency: 0.183 ms.
Write Count: 3490007
Write Latency: 0.044 ms.
Pending Tasks: 0
Key cache capacity: 200000
Key cache size: 10975
Key cache hit rate: 0.9571185054917931
Row cache: disabled
Compacted row minimum size: 179
Compacted row maximum size: 7007506
Compacted row mean size: 4671706
```

8.6 Testing using VisualVM

8.6.1 View of JacORB threads

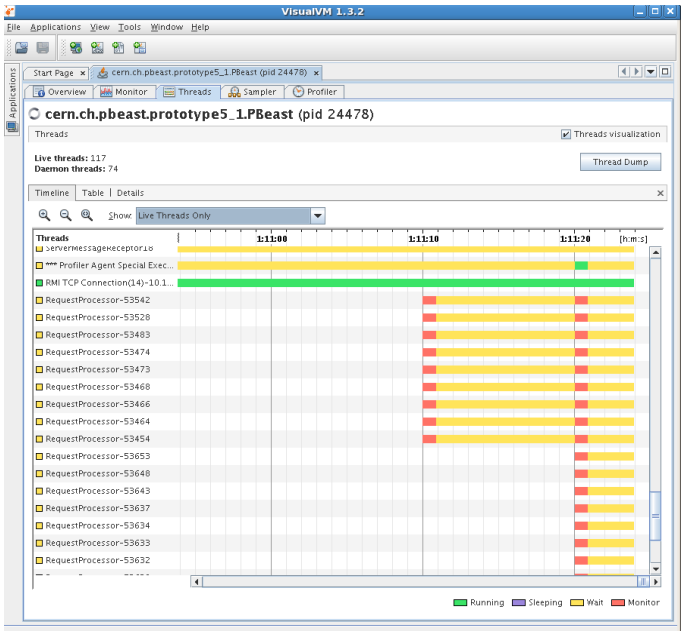


Figure 8.4: Jacorb "RequestProcessor" threads

8.6.2 View of PBEAST's ThreadPoolExecutor threads

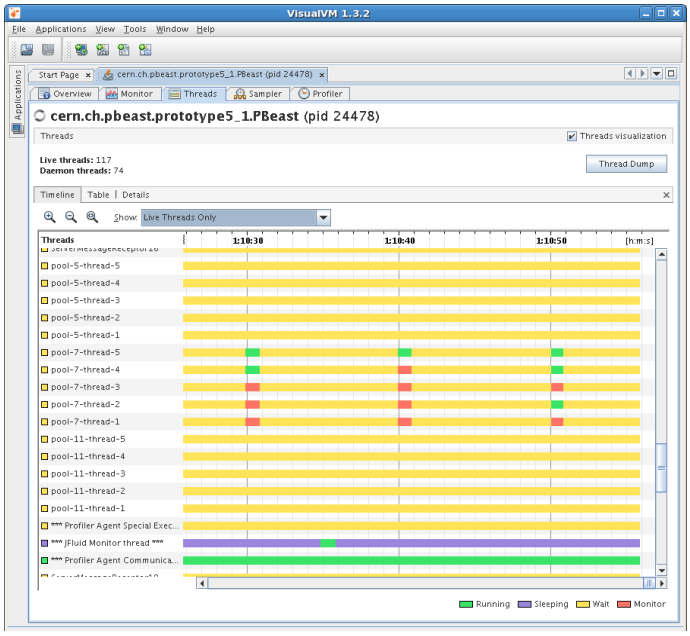


Figure 8.5: Threads in the thread pools of the executor pattern

8.7 Initial Project Gantt Chart

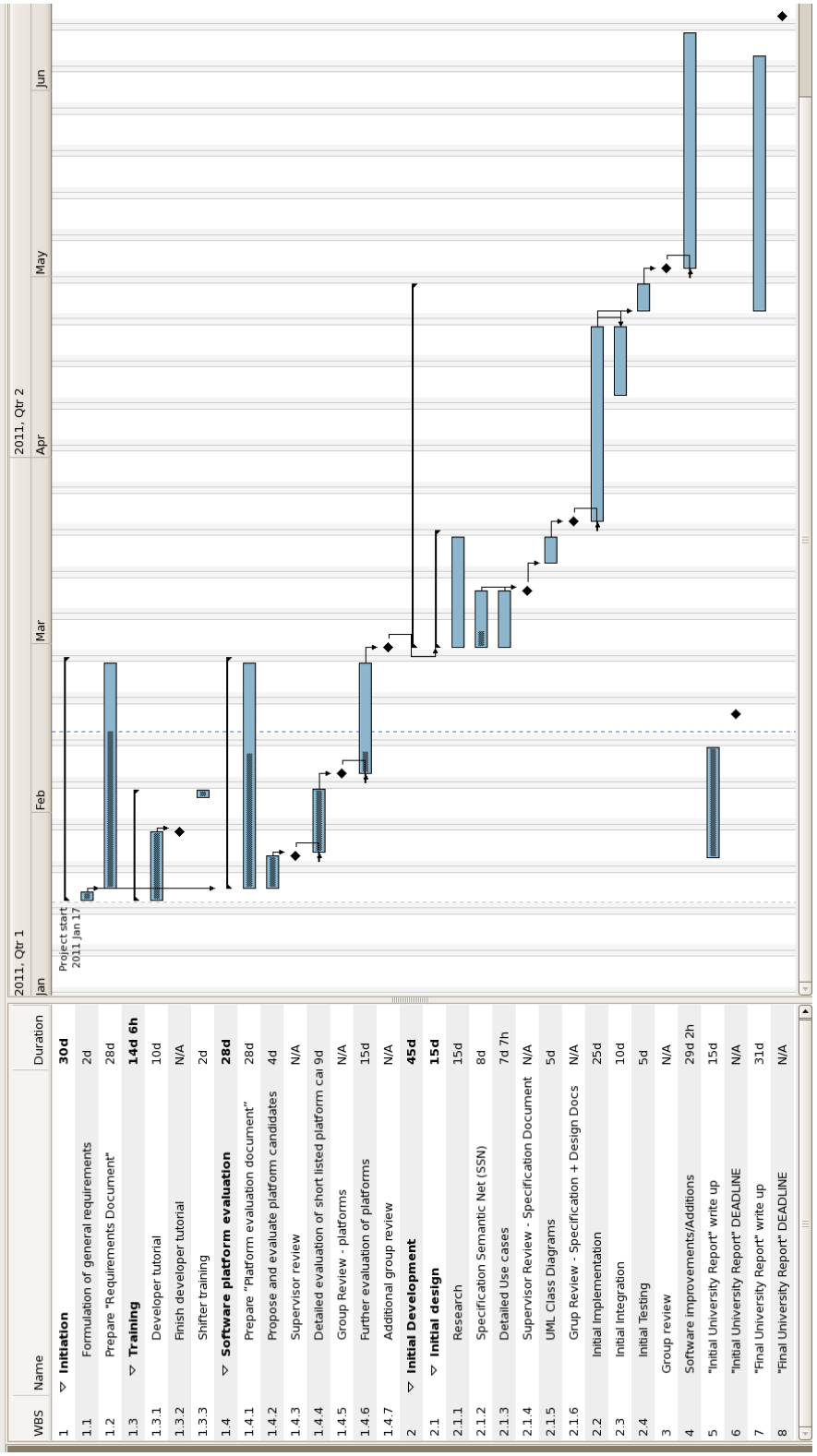


Figure 8.6: Initial Project Gantt Chart

8.8 Project Execution Gantt Chart

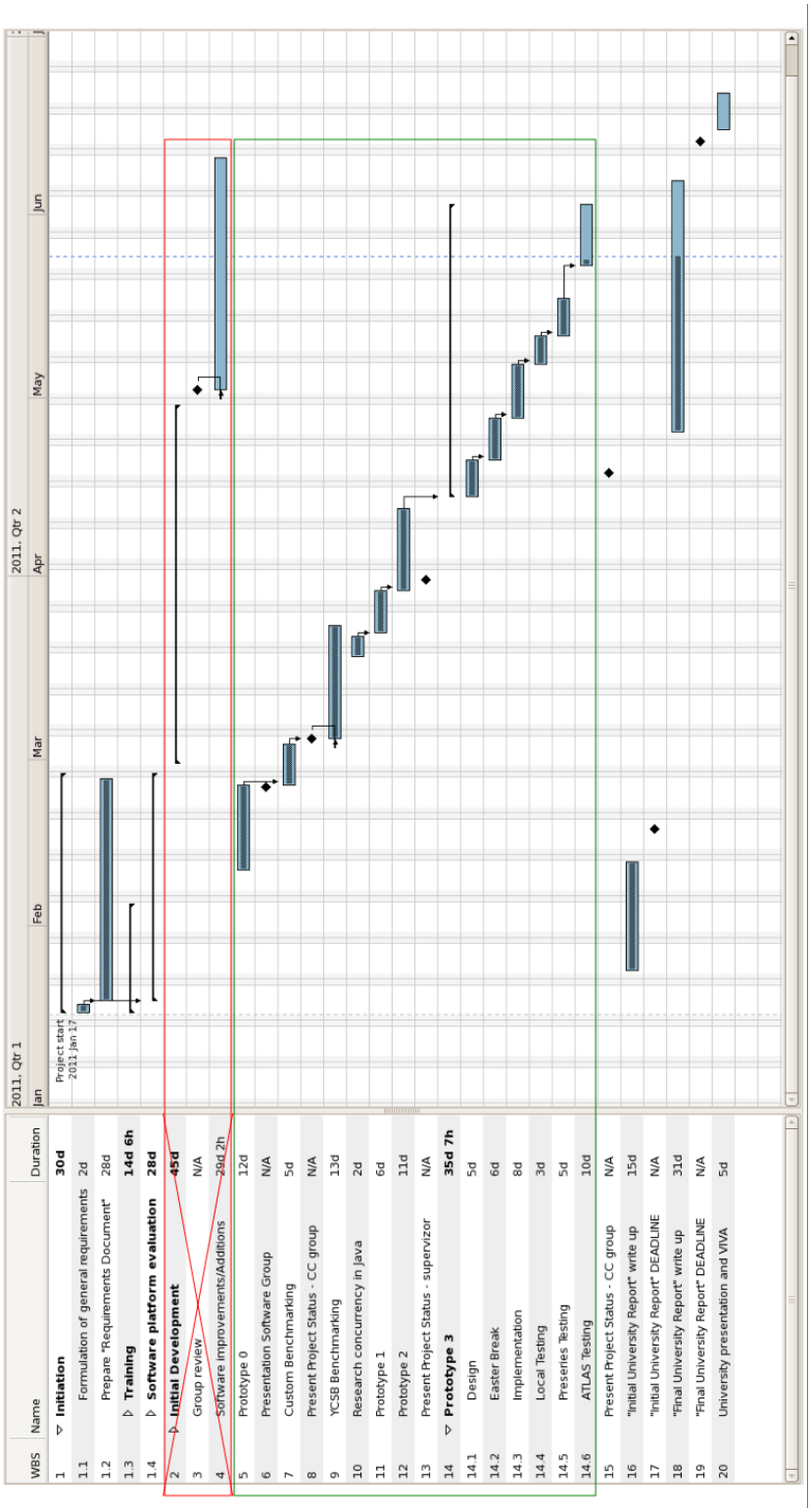


Figure 8.7: Project Execution Gantt Chart

8.9 P BEAST code

8.9.1 ISInfoExtractor Class

```
/**@author Alexandru Dan Sicoe
* PBeast Prototype 3
* Improvements since prototype 2
*       - changes to assure migration from old schema to new schema 1
*       - row splitting
**/
package cern.ch.pbeast.proto3;
import java.io.BufferedReader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Enumeration;
import java.util.HashMap;
import java.util.Vector;
import java.util.concurrent.CompletionService;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ConcurrentMap;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorCompletionService;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import me.prettyprint.hector.api.exceptions.HectorException;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.apache.log4j.Logger;
import org.perf4j.LoggingStopWatch;
import org.perf4j.StopWatch;
import org.perf4j.log4j.Log4JStopWatch;
import cern.ch.pbeast.EmptyArrayException;
import cern.ch.pbeast.EmptyHashMapException;
import dal.AppConfig;
import dal.BaseApplication;
import dal.ComputerProgram;
import ipc.Partition;
import is.AnyInfo; import is.Criteria; import is.EventListener;
import is.InfoDocument; import is.InfoEvent; import is.InfoList;

public class ISInfoExtractor implements EventListener {
        /**@attributes*/
private static final Log log = LogFactory.getLog(ISInfoExtractor5.class);
private static final int NUM_THREADS = 5; //in the executor thread pool
private Vector<String> nodeNames; //name/IPs of all the machines in
//the Cassandra cluster
private Partition partition;//facilitates IPC
private final String isServerName; //name of IS server that this instance
//is responsible for
private final String keyspaceName; //is equal to the partition name
private final String clusterName; //name of Cassandra cluster
private int port; //port on which Cassandra listens on
private final ExecutorService executor; //synchronized task queue with
//pool of worker threads
private AttributeCache attributeCache; //in memory structure that holds
//the last value inserted in the database for each attribute, the current
//row delimiter and the count of values in the current row
private CassandraDBClient cassandraDBClient; //has methods to access database
        /**@constructor*/
public ISInfoExtractor(Partition partition, String isServerName,
String clusterName, int port, Vector<String> nodeNames, boolean debug) {
        this.partition = partition;
        this.isServerName = isServerName;
        this.keyspaceName = partition.getName();
```

```
        this.clusterName = clusterName;
        this.port = port;
        this.nodeNames = nodeNames;
        this.executor = Executors.newFixedThreadPool(NUM_THREADS); //returns an
//obj implementing the ExecutorService interface which is actually a
//ThreadPoolExecutor obj and the following cast is
//safe ((ThreadPoolExecutor)executor).getQueue().size();
        log.debug("Debugging enabled:");
        log.debug("Cluster name="+this.clusterName);
        log.debug("Keyspace name="+this.keyspaceName);
        log.debug("Cluster node names: ");
        for(String s:this.nodeNames) {
                log.debug(s+" ");}
                log.debug("Port="+this.port);
                try {
                        this.cassandraDBClient = new CassandraDBClient(keyspaceName,
clusterName, nodeNames, port, true);
                        log.info("Created Cassandra db client");
                        populateAttributeCache(); //create an in memory cache from
//the Cache metadata column family
                        log.info("Populated in memory attribute cache from the
persistent attribute cache");
                } catch (EmptyArrayException e) {
                        log.error(e.getMessage());
                        e.printStackTrace();
                } catch (HectorException he) {
                        log.error(he.getMessage());
                        he.printStackTrace();
                }
        }

/**Read all values in the "Cache" metadata CF and put them in the
attributeCache**/
private void populateAttributeCache() {
        this.attributeCache = cassandraDBClient.readCacheSchema2(isServerName);
//returns an empty cache object if no values stored in the Cache metadata CF
}

/**Called when an information object is first published in the IS repository**/
public void infoCreated(InfoEvent infoEvent) {
        //servant code executed by the JacORB RequestProcessor threads
        LoggingStopWatch createStopWatch =
new Log4JStopWatch(Logger.getLogger(ISInfoExtractor.class.getCanonicalName()+
".create.TimingLogger")); //gathering statistics about the callback
        createStopWatch.start();
        AnyInfo ai = new AnyInfo();
        LoaderTask task;
        infoEvent.getValue(ai); //does de-marshalling behind the scenes
        try {
                task = new LoaderTask(partition,ai,attributeCache, keyspaceName,
clusterName, nodeNames, port, false); //create task
                executor.execute(task); //submit task for execution
//(place in executor input queue)
        createStopWatch.stop(isServerName);
        } catch (EmptyArrayException e) {
                log.error(e.getMessage());
                e.printStackTrace();
        }
}

        /**Called when an information object is updated in the IS repository**/
public void infoUpdated(InfoEvent infoEvent) {
        //servant code executed by the JacORB RequestProcessor threads
LoggingStopWatch updateStopWatch =
new Log4JStopWatch(Logger.getLogger(ISInfoExtractor.class.getCanonicalName()+
".update.TimingLogger")); //gathering statistics about the callback
        updateStopWatch.start();
        AnyInfo ai = new AnyInfo(); //does de-marshalling behind the scenes
        infoEvent.getValue(ai);
        try {
                LoaderTask task = new LoaderTask(partition,ai,attributeCache,
```

```
keyspaceName, clusterName, nodeNames, port, false); //create task
        executor.execute(task); //submit task for execution (place in executor input queue)
        updateStopWatch.stop(isServerName);
        } catch (EmptyArrayException e) {
                log.error(e.getMessage());
                e.printStackTrace();
        }
}

/**Called when an information object is deleted from the IS repository**/
public void infoDeleted(InfoEvent infoEvent) {
//servant code executed by the JacORB RequestProcessor threads;
//at the moment only gathers statistics
        LoggingStopWatch deleteStopWatch =
new Log4JStopWatch(Logger.getLogger(ISInfoExtractor.class.getCanonicalName()+
".delete.TimingLogger")); //gathering statistics about the callback
        deleteStopWatch.start();
        deleteStopWatch.stop(isServerName);
}

public AttributeCache getAttributeCache() {return attributeCache;}

public long getUpdateCounter() {return updateCounter;}

public String getKeyspaceName() {return keyspaceName;}

/**@return the executor - casted to ThreadPoolExecutor so that the caller
can get the length of the queue, the size of the thread pool etc for
monitoring purposes*/
public ThreadPoolExecutor getExecutor() {return ((ThreadPoolExecutor)executor);}

/**@description Utility method to encapsulate some of the messier exception
handling logic Given in book "JAVA Concurrency in practice", Brian Goetz, p 98
If the Throwable is in Error, throw it; if it is a RuntimeException return it
otherwise throw IllegalStateException*/
public static RuntimeException launderThrowable(Throwable t) {
        if(t instanceof RuntimeException) return (RuntimeException)t;
        else if(t instanceof Error) throw (Error)t;
        else throw new IllegalStateException("Not unchecked",t);
        }
}
```

Bibliography

[1] ATLAS HLT/DAQ/DCS Group. (2003, Jul.), ATLAS high-level trigger, data-acquisition and controls : Technical Design Report. CERN, Sw [Online]. Available:http://cdsweb.cern.ch/record/616089

[2] G. Lehmann Miotto, "Configuration and Control of the ATLAS Trigger and Data Acquisition", Nuclear Instruments and Methods in Physics Research A 623, pp.549–551, Mar. 2010

[3] CASTOR. Main page for the CERN Advanced Storage Manager. [Online] Viewed 2011 January. Available: http://castor.web.cern.ch/castor/

[4] ADAM web interface. Main CERN twiki project page. [Online] Viewed 2011 May. Available: http://twiki.cern.ch/twiki/bin/viewauth/Atlas/AdamComponent

[5] Linux SLC5. Main Scientific Linux CERN page. [Online] Viewed 2011 January. Available: http://linux.web.cern.ch/linux/scientific5/

[6] Agile process. Iterative Planning Page. [Online]. Viewed 2011 April. Available: http://www.agile-process.org/iterative.html

[7] J. F. Gantz, "The Expanding Digital Universe - A Forecast of Worldwide Information Growth Through 2010", an IDC White Paper - sponsored by EMC, March 2007

[8] Microsoft Research, "The Fourth Paradigm, Data-Intensive Scientific Discovery", ISBN 978-0-9825442-04, 2009

[9] D. Savu. (2011, Jan.), Integrated System for Performance Monitoring of the ATLAS TDAQ Network [Online], Available:http://cdsweb.cern.ch/record/1322435?ln=en

[10] Yahoo! Research, "Benchmarking Cloud Serving Systems with YCSB", Proceedings of the 1st ACM symposium on Cloud computing, p.143-154, 2010

[11] Wikipedia. Entity-attribute-value Model Home Page. [Online]. Viewed 2011 January. Available: http://en.wikipedia.org/wiki/Entity-attribute-value_model

[12] Wikipedia. NoSQL Home Page. [Online]. Viewed 2011 January. Available: http://en.wikipedia.org/wiki/NoSQL

[13] Google Labs. (2006, Nov.), Bigtable: A Distributed Storage System for Structured Data. Google, Cal [Online]. Available:http://labs.google.com/papers/bigtable.html

[14] The Apache Software Foundation. Apache Cassandra Home Page. [Online]. Viewed 2011 February. Available: http://cassandra.apache.org/

[15] Cassandra Wiki. Architecture Overview Page. [Online]. Viewed 2011 February. Available: http://wiki.apache.org/cassandra/ArchitectureOverview

[16] E. Hewitt, "Cassandra: The Definitive Guide", O'Reilly, ISBN: 978-1-449-39041-9

[17] SEDA. Stage Event Driven Architecture Home Page. [Online]. Viewed March 2011. Available: http://www.eecs.harvard.edu/~mdw/proj/seda/

[18] The Apache Software Foundation. Thrift Home Page. [Online]. Viewed March 2011. Available: http://thrift.apache.org/

[19] Prettyprint. Hector Java API to Cassandra Home Page. [Online]. Viewed April 2011. Available: http://prettyprint.me/2010/08/06/hector-api-v2/

[20] Wikipedia. MD5 home page. [Online]. Viewed June 2011. Available: http://prettyprint.me/2010/08/06/hector-api-v2/

[21] 10gen. MongoDB Home Page. [Online]. Viewed 2011 February. Available: http://www.mongodb.org/

[22] National Centre for Supercomputing Applications (NCSA). HDF5 Home Page. [Online]. Viewed 2011 February. Available: http://www.hdfgroup.org/HDF5/

[23] The Apache Software Foundation. HBase Home Page. [Online]. Viewed 2011 February. Available: http://cassandra.apache.org/

[24] Cloudkick. Cassandra blog page. [Online]. Viewed 2011 February. Available: http://www.cloudkick.com/blog/2010/mar/02/4_months_with_cassandra/

[25] Object Management Group. CORBA website. [Online]. Viewed 2011 May. Available: http://www.corba.org/

[26] Jacorb Home Page. [Online]. Viewed 2011 March. Available: http://www.jacorb.org/

[27] Jacorb Programming Guide PDF Document. [Online]. Viewed 2011 March. Available: http://www.jacorb.org/releases/2.3.0/ProgrammingGuide.zip, p.39-48, p.53, p.137-141

[28] B. Goetz, "Java concurrency in practice", Adison Wesley, ISBN 0-321-34960-1

[29] Wikipedia. Memory model in computing home page. [Online]. Viewed 2011 March. Available: http://en.wikipedia.org/wiki/Memory_model_(computing)

[30] Java Language Specification, Third Edition. Chapter 17 Threads and Locks. [Online]. Viewed 2011 March. Available: http://java.sun.com/docs/books/jls/third_edition/html/memory.html

[31] Java. Unbounded queues. [Online]. Viewed 2011 March. Available: http://download.oracle.com/javase/6/docs/api/

[32] Launchpad. Byobu home page. [Online]. Viewed 2011 April. Available: https://launchpad.net/byobu

[33] Apache. Log4j home page. [Online]. Viewed 2011 May. Available: http://logging.apache.org/log4j/

[34] Codehouse. Perf4j home page. [Online]. Viewed 2011 May. Available: http://perf4j.codehaus.org/

[35] VisualVM. Homepage. [Online]. Viewed 2011 May. Available: http://visualvm.java.net/

[36] Top Linux command manual page. [Online]. Viewed 2011 March. Available: http://linuxcommand.org/man_pages/top1.html

[37] Iostat Linux command manual page. [Online]. Viewed 2011 March. Available: http://linuxcommand.org/man_pages/iostat1.html