Log Based Method for Faster IoT Queries

by

Anubha Jain
201511029

A Thesis Submitted in Partial Fulfilment of the Requirements for the Degree of

MASTER OF TECHNOLOGY
in
INFORMATION AND COMMUNICATION TECHNOLOGY
to

DHIRUBHAI AMBANI INSTITUTE OF INFORMATION AND COMMUNICATION TECHNOLOGY

June, 2017

Declaration

I hereby declare that
(i) the thesis comprises of my original work towards the degree of Master of Technology in Information and Communication Technology at DA-IICT and has not been submitted elsewhere for a degree,
(ii) due acknowledgement has been made in the text to all the reference material used.

Signature of Student

Certificate

This is to certify that the thesis work entitled "Log Based Method for Faster IoT Queries" has been carried out by Anubha Jain (201511029) for the degree of Master of Technology in Information and Communication Technology at Dhirubhai Ambani Institute of Information and Communication Technology under my/our supervision.

Dr. Minal Bhise
Thesis Supervisor

Acknowledgements

The work presented here is a result of support from many people. I would like to extend my deepest regards to all those who have helped me in this journey.

Primarily, it is the encouragement and constant guidance of my thesis advisor Dr. Minal Bhise which has kept me focused to carry out this research work. I would like to take this opportunity to thank her for dedicating her time and efforts for this work. Her valuable suggestions have kept me in correct direction to complete this work.

A special gratitude goes to Trupti Di and Zubain for helping me with many resources to carry out the research. Our discussions have helped me a lot to gain insights about the work.

I am also thankful to members of my thesis examination panel who have always given meaningful standpoints about my work. Thanks for very generously pointing and many a times ignoring my mistakes.

Last but not the least, thanks to the lab staff of Dhirubhai Ambani Institute of Information and Communication Technology (DA-IICT), Gandhinagar, for providing necessary facilities and environment to carry out the research work.

Table of Contents

Abstract

With increase in the size of Internet of Things IoT device networks and applications, tremendous increase is witnessed in the size of IoT data. To build smart applications using IoT data, its efficient storage is important to facilitate faster queries. IoT data is represented in Resource Description Framework RDF and stored in relational format. This thesis addresses the issue of faster query processing of this data. It presents a Log Based Method LBM to partition IoT data. IoT systems exhibit skewedness in data access patterns as some records are accessed more frequently than the other. LBM exploits this skewedness in access patterns of records. It incorporates Forward Algorithm FA and Backward Algorithm BA to analyse the query workload and partition the basic triple table into hot and cold data tables. For our experiment, 8% of hot data table is found to solve 78.6% queries. The query execution is found to be 67.5% faster on partitioned data than triple table. To further accelerate FA and BA, we have executed them in parallel as well which is found to be 42% faster than its serial execution.

Keywords: Backward Algorithm; data partitioning; Exponential Smoothing; Forward Algorithm; Hot data; IoT; Log Based Method; QET; RDF

List of Principal Symbols and Acronyms

α	Alpha

Other symbols are defined at first occurrence; where necessary some symbols are redefined in the text.

IoT	Internet of Things
ES	Exponential Smoothing
BA	Backward Algorithm
FA	Forward Algorithm
QET	Query Execution Time
OLTP	Online Transaction Processing
RDF	Resource Description Framework
WAHP	Workload Aware Hybrid Partitioning

List of Tables

List of Figures

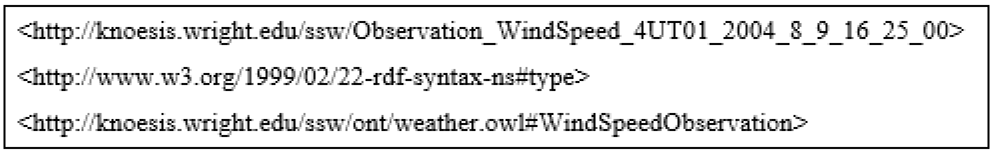
Chapter 1

Introduction

1.1 General

Internet of Things IoT aims at providing better user experience by effective management of user resources. The resources can range from personal devices like smart home automation system, wearables, cars, etc. to public resources like transportation, traffic, water, energy, etc. It involves analysis of usage, prediction of possible issues and providing ways for optimal usage of the resources. These decisions are taken by analysis of data integrated by real time monitoring of these resources. Internet of Things IoT is an interconnection of these monitoring devices referred to as things. It is a grid of sensors which collect data from these resources. Data collected from one resource can be exchanged with other resource and can be used to build smart applications. e.g. an application can be developed to assist farmers to take informed decisions by combining weather data collected by sensors along with other geographical data of that location.

To represent data, a standard format is required so that it can be exchanged and mapped across applications to carry out sophisticated tasks for users. Resource Description Framework RDF is one such standard format which is a world-wide accepted way for description of resources, concepts and relationship between them [1]. In this format, the information about a resource is represented using triple. A triple is an expression of the form subject-property-object. Here, subject is a web resource, property is a trait of the resource and object is the value of that property. Their values are represented in the form of Universal Resource Identifiers URIs. It ensures that each concept is tied to a unique definition and can be linked across domains using these identifiers. In an IoT triple as shown in Fig. 1.1, "http://knoesis.wright.edu/ssw/Observation_WindSpeed_4UT01_2004_8_9_16_25_00" is the URI of subject which is an observation made by an IoT sensor. "http://www.w3.org/1999/02/22-rdf-syntax-ns#type" is the URI of the property 'type' for the subject and "http://knoesis.wright.edu/ssw/ont/weather.owl#WindSpeedObservation" is the URI for the object.



```
<http://knoesis.wright.edu/ssw/Observation_WindSpeed_4UT01_2004_8_9_16_25_00>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://knoesis.wright.edu/ssw/ont/weather.owl#WindSpeedObservation>
```

Fig. 1.1 Example of IoT Triple

IoT data can be stored as a set of triples or as XML documents or as a graph of triples. Representation of IoT data in XML statements makes it machine readable. If an application receives data from various domains, same format of data representation enables establish connection between them. It allows prompt information retrieval between domains and can help solving complex queries. It can be used to build sophisticated applications for user to provide them better functionalities and experience.

To provide better features with existing products and to offer new products, the usage of IoT devices are rapidly increasing. According to Cisco Visual Networking Index [2], over the next five years, global IP networks will support up to 10 billion new devices and connections, increasing from 16.3 billion in 2015 to 26.3 billion by 2020. The size of Linked Observation Data LOD used in this thesis is 114M triples. Many applications on internet other than IoT like healthcare, life sciences, geo sensing are also using RDF format to represent their data. The size of data handled by such systems is huge e.g. DBLP, a computer science bibliography dataset contains 286M triples, and Wikidata contains 2262M triples. To build smart applications using these datasets, we need faster query processing rate. One of the way to achieve it is the efficient storage of data which should be generic and scale to a large amount of data. For this thesis, IoT data is stored as a set of triples in a relational database. All triples are stored in a three column (subject, property, and object) schema. Each triple is stored as a separate row in the table. It is a very general approach and any kind of IoT data can be stored in this form. But there are many issues when storing data this way. Like, to process any query on this data, a few self-joins may be required. The space required for the self-join operation on a huge table may exceed the size of memory available. The queries can also become quite slow which decreases the efficiency of any system in terms of query execution time. This solution may not scale well if the number of triples increase. Thus, storing IoT dataset in a single triple table is quite infeasible. A better solution is to partition this table. To answer any query in the partitioned system, we may require only some of the data sections and not others. Though this approach will also require joins, but as the table size is smaller than the original table, joins are feasible. This approach takes less time to answer queries as the size of data scanned is less.

There are many ways to partition IoT dataset into multiple tables. Some of the partitioning techniques are discussed in Chapter 2. It can be partitioned by either data aware partitioning method or query aware partitioning method. The former considers the type of data and clusters similar data in same partition. The later method is based upon the query workload handled by the system and is used in this thesis.

If the workload is analysed for any IoT data based system, we observe a discrepancy in the access pattern of its records. For an observation period, some data is queried more frequently than the other. Some data may not be accessed at all. This asymmetry in data access pattern leads to data skewedness. It can be natural skewedness which means that some data items are preferred over other e.g. from a variety of online courses, some courses are more popular among users than the other courses. Similarly, in everyday processing in a banking application, transaction details of users are more important than address and contact details of user. Another reason for skewedness can be data ageing. When some recent records in dataset are queried more often than the older entries then there is skewedness based upon the age. As the age of rows increase, their chance of access decrease for systems showing data ageing in access patterns e.g. to forecast real-time weather, only recent weather data is required. Similarly, in a courier delivery system, the package information is expected to be accessed till its delivery and is rarely accessed afterwards. So, if data access is analysed in any system, patterns of natural skewedness and data ageing can be identified in it.

This thesis presents partitioning of IoT data in frequently used data and rarely used data by exploiting patterns of data skewedness. Frequently accessed records can be identified as described in chapter 3. The data which is highly queried is called as hot data. Rest of the data is called cold data which is infrequently or not queried at all. Records which have access frequency in between hot and cold can be considered as warm data. As hot data can solve most of the queries, there is no need to search the entire table every time. This saves the time for searching the complete dataset every time for a query. Hot and cold data are stored in separate tables. So, the objective of this thesis is to accurately identify the hot and cold data from a database storing IoT data so that the queries can be answered quickly. It is done by analysing the query pattern of the application for a certain time. In a real-world application, the workload is dynamic and changes with its usage. To make this work applicable to accommodate these changes, the identification algorithms can be re-run frequently. The frequency of run depends on algorithm complexity and system characteristics.

Once the hot data is accurately identified, it can also be moved in the main memory. The motivation behind this is the very high-speed access of data from main memory. In a traditional Disk Residing Database System DRDB, data is kept on the disk. Its retrieval involves many CPU instructions and use of several buffers which takes higher seek time for querying data as compared to main memory. The Main Memory Database Systems MMDB stores data in main memory. Querying data from such system requires less number of CPU instructions thus response time is less in MMDB as compared to DRDB. Hot data which is frequently accessed is very less in size than the complete dataset and can be stored in main memory while the voluminous cold data continues to stay on disk. This provides better response time of our system as compared to when entire data is stored on disk.

In this thesis, a method to accurately identify hot and cold sections from an IoT dataset is presented. It is done by maintaining log of record accesses made by the system workload. This log is analysed and the record accesses are estimated from it. The records with highest estimated access frequencies are considered as hot records. The remaining records with lower estimated access frequencies are considered as cold records. The queries are then run on the partitioned tables and the query performance in terms of query execution time is found better than the triple table.

1.2 Objective

The objective of this thesis is to:

i. Partition IoT dataset by identifying hot and cold data.
ii. Improve the query performance in terms of query execution time of workload on IoT dataset.

Some assumptions are made when implementing experiments in this thesis. The IoT data used is static and is stored in relational format. The workload is also static with known queries and their ranking.

1.3 Organisation of Thesis

In this thesis, Literature survey is presented in Chapter 2. The method which is used to partition IoT dataset is explained in Chapter 3. Then the experimental flow is given in Chapter 4. The results are discussed in Chapter 5 followed by conclusion in Chapter 6.

1.4 Contribution

The contributions of the work conducted during this thesis are:

i. IoT dataset is successfully partitioned in hot and cold data sections.
ii. The query performance has improved as the query execution time has decreased.
iii. Small portion of hot data is identified which can answer most of the queries.
iv. The technique is general and can also be used for any other RDF dataset.

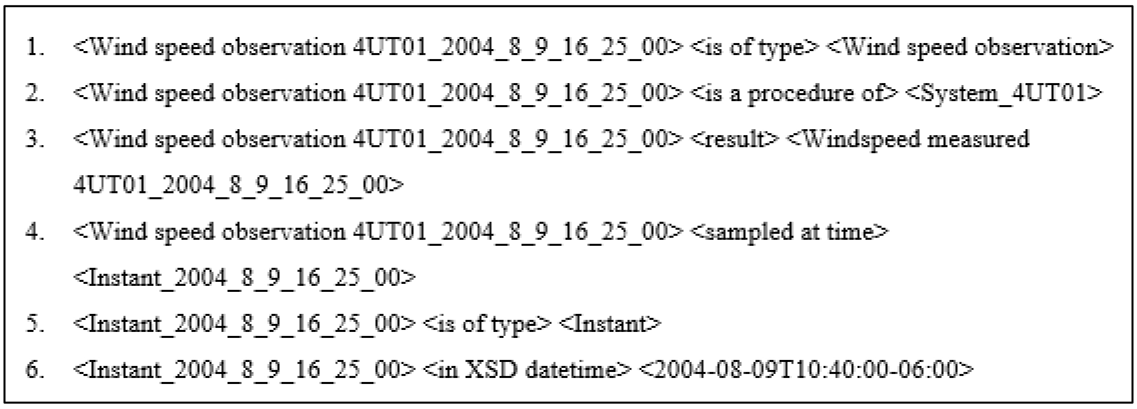
Chapter 2

Literature Survey

The chapter includes discussion on methods that can be used to store IoT data in relational format. It then discusses the requirement to partition IoT data and presents several works done to partition it.

2.1 Representation of IoT Data

RDF has been accepted as a standard format to represent IoT data. It provides standard libraries and formats so that data of a system can be mapped and connected to the data of other systems. It allows exchange of data among devices and to run distributed queries on IoT data. The interconnection of data from different domains can be used to build smart applications. IoT data can be represented in XML format or as a directed labelled graph. Consider an example of IoT statements as shown in Fig. 2.1:

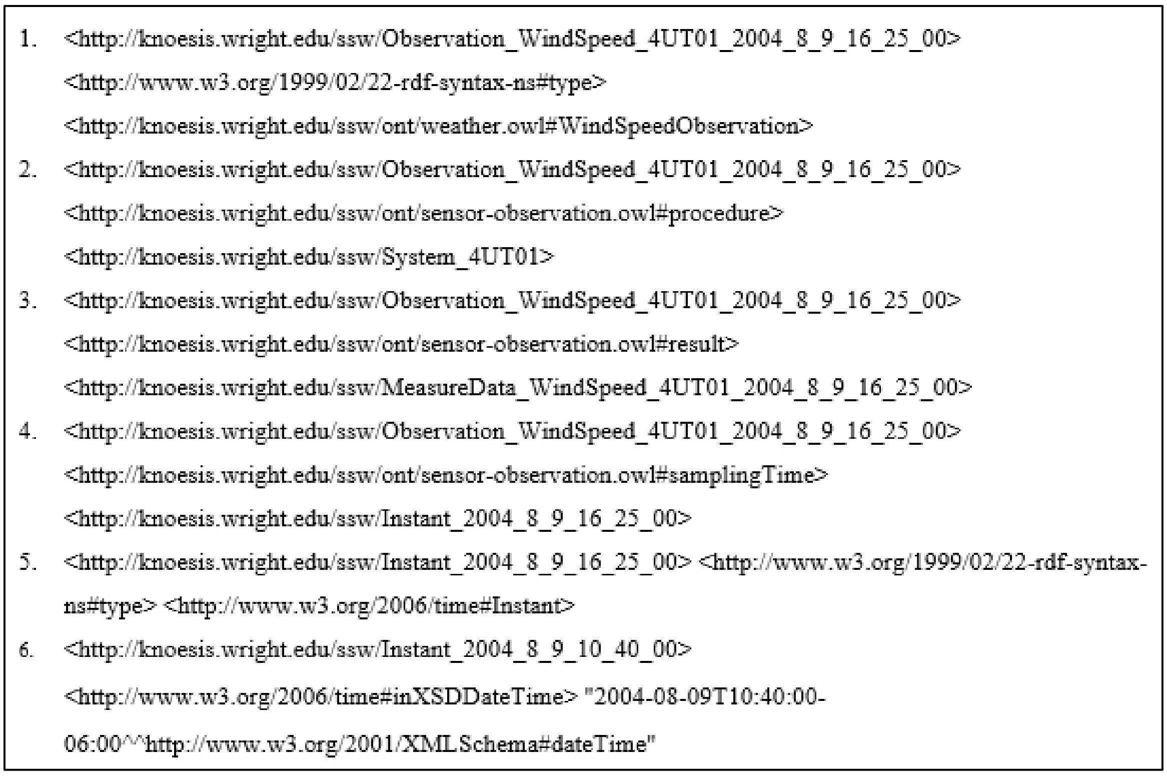


1. <Wind speed observation 4UT01_2004_8_9_16_25_00> <is of type> <Wind speed observation>
2. <Wind speed observation 4UT01_2004_8_9_16_25_00> <is a procedure of> <System_4UT01>
3. <Wind speed observation 4UT01_2004_8_9_16_25_00> <result> <Windspeed measured 4UT01_2004_8_9_16_25_00>
4. <Wind speed observation 4UT01_2004_8_9_16_25_00> <sampled at time> <Instant_2004_8_9_16_25_00>
5. <Instant_2004_8_9_16_25_00> <is of type> <Instant>
6. <Instant_2004_8_9_16_25_00> <in XSD datetime> <2004-08-09T10:40:00-06:00>

Fig. 2.1 IoT Triples

2.1.1 N-Triples

IoT triples can be represented in N-Triples format as shown in Fig. 2.2:



```
1.  <http://knoesis.wright.edu/ssw/Observation_WindSpeed_4UT01_2004_8_9_16_25_00>
    <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    <http://knoesis.wright.edu/ssw/ont/weather.owl#WindSpeedObservation>
2.  <http://knoesis.wright.edu/ssw/Observation_WindSpeed_4UT01_2004_8_9_16_25_00>
    <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#procedure>
    <http://knoesis.wright.edu/ssw/System_4UT01>
3.  <http://knoesis.wright.edu/ssw/Observation_WindSpeed_4UT01_2004_8_9_16_25_00>
    <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#result>
    <http://knoesis.wright.edu/ssw/MeasureData_WindSpeed_4UT01_2004_8_9_16_25_00>
4.  <http://knoesis.wright.edu/ssw/Observation_WindSpeed_4UT01_2004_8_9_16_25_00>
    <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#samplingTime>
    <http://knoesis.wright.edu/ssw/Instant_2004_8_9_16_25_00>
5.  <http://knoesis.wright.edu/ssw/Instant_2004_8_9_16_25_00> <http://www.w3.org/1999/02/22-rdf-syntax-
    ns#type> <http://www.w3.org/2006/time#Instant>
6.  <http://knoesis.wright.edu/ssw/Instant_2004_8_9_10_40_00>
    <http://www.w3.org/2006/time#inXSDDateTime> "2004-08-09T10:40:00-
    06:00^^http://www.w3.org/2001/XMLSchema#dateTime"
```

Fig. 2.2 IoT Triples in N-Triples Format

2.1.2 Relational Format

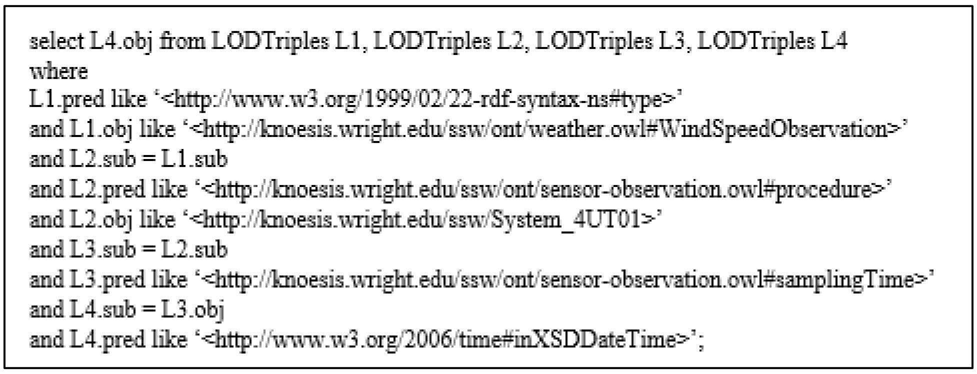
IoT triples can be stored in a relational, non-relational database or other data sources. Of many available options, one general way to store IoT data is in relational database format. It can be stored in a table of three column (subject, property and object) schema called triple table. Each triple can be stored as a row in the tab as shown in Table 2.1. It is a very easy and general approach to implement. The values of subject and property columns are URIs and an object is either URI or a string literal. For better space management and quick search, instead of storing long strings in the triple table, they can be stored in a separate table. The encoded shorter versions or keys are stored in triple table.

Table 2.1 IoT Triples in Relational Format

Subject	**Property**	**Object**
<http://knoesis.wright.edu/ssw/Observation_WindSpeed_4UT01_2004_8_9_16_25_00>	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://knoesis.wright.edu/ssw/ont/weather.owl#WindSpeedObservation>

<http://knoesis.wright.edu/ssw/Observation_WindSpeed_4UT01_2004_8_9_16_25_00>	<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#procedure>	<http://knoesis.wright.edu/ssw/System_4UT01>
<http://knoesis.wright.edu/ssw/Observation_WindSpeed_4UT01_2004_8_9_16_25_00>	<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#result>	<http://knoesis.wright.edu/ssw/MeasureData_WindSpeed_4UT01_2004_8_9_16_25_00>
<http://knoesis.wright.edu/ssw/Observation_WindSpeed_4UT01_2004_8_9_16_25_00>	<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#samplingTime>	<http://knoesis.wright.edu/ssw/Instant_2004_8_9_16_25_00>
<http://knoesis.wright.edu/ssw/Instant_2004_8_9_16_25_00>	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://www.w3.org/2006/time#Instant>
<http://knoesis.wright.edu/ssw/Instant_2004_8_9_16_25_00>	<http://www.w3.org/2006/time#inXSDDateTime>	"2004-08-09T10:40:00-06:00^^http://www.w3.org/2001/XMLSchema#dateTime"

From this table, if we want to find the time at which wind speed is measured at System_4UT01, then we have to perform 4 self-joins on the table. The SQL query to get the required result is as shown in Fig. 2.3:



```
select L4.obj from LODTriples L1, LODTriples L2, LODTriples L3, LODTriples L4
where
L1.pred like '<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>'
and L1.obj like '<http://knoesis.wright.edu/ssw/ont/weather.owl#WindSpeedObservation>'
and L2.sub = L1.sub
and L2.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#procedure>'
and L2.obj like '<http://knoesis.wright.edu/ssw/System_4UT01>'
and L3.sub = L2.sub
and L3.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#samplingTime>'
and L4.sub = L3.obj
and L4.pred like '<http://www.w3.org/2006/time#inXSDDateTime>';
```

Fig. 2.3 SQL Query to Fetch Data from IoT Dataset

2.2 Partitioning of IoT Data

To solve a simple query, it requires a number of self-joins of the table. The queries in real world application are more complex and they require more self-joins of the triple table. Depending on the application, the table size can be millions of triples, on which imposing self joins requires large memory. The memory requirement can even exceed the available space which makes the query execution impractical. The system developed using this data storage approach is quite slow and is practically infeasible. To build faster applications, a better way is required to store IoT data which should perform well in case of scaling of system.

Instead of storing all data in single table, data can be partitioned and stored in a number of tables. A technique is required for data partitioning and distribution across these tables. The partitioning of IoT data can be done using the following two approaches:

2.2.1 Data Centric Partitioning Approach

The partitioning of IOT data is based upon the values of subject and property. For these approaches, an analysis of the dataset is required. Some data-centric partitioning approaches are:

2.2.1.1 Property Table Partitioning

It is based upon the value of properties in dataset [3]. There are two ways in which it can be done:

2.2.1.1.1 Clustered Property Table Partitioning

Properties that tend to be defined together for the subjects are clustered together. Consider the following set of triples stored in a triple table:

Table 2.2 Triple Table

Subject	Property	Object
ID1	Type	Teacher
ID2	Type	Student
ID3	Type	Student
ID4	Type	Teacher
ID5	Type	Student
ID1	Name	John
ID2	Name	Victoria

ID3	Name	Smith
ID4	Name	Julio
ID6	Name	Joe
ID1	Teaches	Maths
ID1	Teaches	Science
ID4	Teaches	English
ID2	Studies	Maths
ID3	Studies	Science
ID1	Degree	M.S.
ID4	Degree	B.A.

To create a clustered property table for the above example, we analyse the distinct property values. Properties like type and name are defined for similar subjects. Thus, a property table with columns- subject, type and name is created. It contains subjects from triple table which have type and name as their properties. More than one property table can be created with a specific property appearing in at most one table. The remaining triples are stored in a left-over triple table as follows:

Table 2.3 Clustered Property Table

Subject	**Type**	**Name**
ID1	Teacher	John
ID2	Student	Victoria
ID3	Student	Smith
ID4	Teacher	Julio
ID5	Student	NULL
ID6	NULL	Joe

Table 2.4 Left-over Triple Table

Subject	**Property**	**Object**
ID1	Teaches	Maths
ID1	Teaches	Science
ID4	Teaches	English
ID2	Studies	Maths
ID3	Studies	Science
ID1	Degree	M.S.
ID4	Degree	B.A.

2.2.1.1.2 Property Class Table Partitioning

Triples of same value of property 'type' are clustered in a class. A table is created for each class to store values of other properties of all its subjects. Triples which cannot be related to any class are stored in a left-over triple table. In the example, we create 'Student' and 'Teacher' classes and partition table as follows:

Table 2.5 Student Class Table

Subject	Name	Studies
ID2	Victoria	Maths
ID3	Smith	Science
ID5	NULL	NULL

Table 2.6 Teacher Class Table

Subject	Name	Teaches	Degree
ID1	John	Math	M.S.
ID4	Julio	English	B.A.

Table 2.7 Left-over Triple Table

Subject	Property	Object
ID6	Name	Joe

Advantages

- It is easier to solve any query based upon a subject value when table is partitioned using any of these two approaches. The subject-subject self-joins are reduced as most of the properties defined for a subject are stored in the same table. But they cannot be completely avoided e.g. in clustered property table approach, to find names of teachers who teach maths requires a join between the two property tables.

Disadvantages

- Queries which are not based on the type-property require more joins in clustered property table.
- Queries which are based on unspecified property types require more joins in case of property-class tables.
- It also includes NULL values in the tables as not all subjects will have values of all the properties in the table. If we try to include more properties in a table, then the number of NULL values will increase and it will create sparse tables. If we include less number of properties in a table then the table is less

sparse but the number of joins to solve a query will increase. Depending upon the application, tables are partitioned in a way so that they are not too sparse and require less number of joins to reduce query complexity.

- As subject is the primary key, a subject having multiple values defined for a property cannot be stored in this table e.g. in Teacher class table, ID1 also teaches science, but the information cannot be stored as it will require multiple rows.
- According to the application, a special algorithm needs to be developed to analyse data and create a proper schema. It makes the method less generic and it is not widely used.

2.2.1.2 Vertical Partitioning

This approach divides the triple table into n binary tables where n is the number of unique property values in the dataset [3]. For each property, a binary table (columns: subject, object) is formed. Each row stores the subject for which that property is defined and the value of that property. In our example, following binary tables will be created:

Table 2.8 Type

Subject	**Object**
ID1	Teacher
ID2	Student
ID3	Student
ID4	Teacher
ID5	Student

Table 2.9 Name

Subject	**Object**
ID1	John
ID2	Victoria
ID3	Smith
ID4	Julio
ID6	Joe

Table 2.10 Teaches

Subject	**Object**
ID1	Maths
ID1	Science

ID4	English

Table 2.11 Studies

Subject	Object
ID2	Maths
ID3	Science

Table 2.12 Degree

Subject	Object
ID1	M.S.
ID4	B.A.

Advantages

- It supports multi-valued attributes. If for a subject there are multiple objects for a property, they can be stored as separate rows in the binary table.
- No NULL values are stored as the subject for which a property is not defined is not stored at all.
- No special algorithm based on the type and application of data is required for partitioning.

Disadvantage

- Joins between tables are required to solve queries which need information of various properties for a subject.

2.2.1.3 Data Centric RDF Storage

To eliminate the requirement of joins between the tables, the technique aims at storing maximum RDF data in a table [4]. The approach is carried in two phases: clustering phase in which sets of correlated properties are identified. If the degree of correlation is more than a threshold value, a n-ary property table is formed of properties which are defined for a large number of subjects. The data defined for remaining properties are stored in binary tables. A high value of threshold results in a highly correlated cluster of properties whereas a low value results in more clusters which are loosely correlated. At the end of this phase, a property can exist in more than one cluster. In the partitioning phase, the clusters are made non-overlapping and all clusters have null values below a threshold.

Reducing number of null values ensure efficient storage and faster queries. As the properties which are correlated have more chances to be queries together, this partitioning technique reduces the number of joins required to solve any query. It results in faster query processing with less space requirements.

A similar technique is implemented in [5] to partition an IoT dataset represented in RDF format based upon its type of data.

2.2.2 Query Centric Partitioning Approaches

The partitioning of IOT data is based on the access pattern of the data. It requires analysis of system workload. It essentially focuses to cluster data which is queried often. Thus, the queries require less number of joins and take less time. Following are some query centric partitioning approaches:

2.2.2.1 Efficient and Adaptable Query Workload Aware Management for RDF Data

This partitioning technique for RDF data is based upon the different type of query workload on the system [6]. It analyses an initial query workload and find out the properties that are frequently queried together. Related properties are clustered together in a fragment such that each property is part of only one fragment. Each fragment is stored as n-ary tables while the triples of remaining properties are stored in a triple table. Any further changes in the workload pattern are monitored to determine any subsequent change required in the property table schema. The process is done in following phases:

a) Vertical Partitioning Phase: In this phase, group of properties which are closely related according to the workload are found. First step is to build an Attribute Usage Matrix AUM in which each row represents a query and each column represents a property. An entry in the matrix is 1 if the query accesses that property or is 0 otherwise.

$$AUM = \left\{ \begin{array}{c|ccccc} & a_1 & a_2 & a_3 & a_4 & \cdots \\ \hline Q_1 & 1 & 1 & 0 & 0 & \cdots \\ Q_2 & 0 & 1 & 1 & 0 & \cdots \\ Q_3 & 0 & 0 & 1 & 1 & \cdots \\ Q_4 & 1 & 1 & 1 & 0 & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \end{array} \right\} \qquad AAM = \left\{ \begin{array}{c|ccccc} & a_1 & a_2 & a_3 & a_4 & \cdots \\ \hline a_1 & 30 & 10 & 0 & 5 & \cdots \\ a_2 & 10 & 40 & 18 & 4 & \cdots \\ a_3 & 0 & 18 & 32 & 7 & \cdots \\ a_4 & 5 & 4 & 7 & 42 & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \end{array} \right\}$$

Fig. 2.4 Attribute Usage Matrix and Attribute Affinity Matrix

Second step is to create an Attribute Affinity Matrix AAM in which each row and column represents the unique properties. An entry in the matrix is the number of queries that simultaneously access both properties. Higher value of affinity among two properties indicate more probability of them being in same fragment. Each diagonal entry is the total number of times a property is accessed by any query. A lower value indicates less access and thus more chances of it remaining in the general triple table. Top K attributes from the AAM with highest total access value are selected and remaining attributes are removed from the matrix. Any property with total access number less than a user defines threshold N is also removed from the matrix.

Clustering: From the modified Attribute Affinity Matrix, we now find and keep properties with high affinities for each other in same fragment. Remaining properties are kept in a separate fragment. It surrounds large values of matrix with large values and small one with small values. It results in formation of a Clustered Affinity Matrix CAM.

$$CAM = \left\{ \begin{array}{c|ccccccc|ccc} & \multicolumn{7}{c|}{\text{Upper}} & \multicolumn{3}{c}{\text{Lower}} \\ & a_6 & a_2 & a_3 & a_7 & a_{10} & a_8 & a_1 & a_9 & a_4 & a_5 \\ a_6 & 90 & 30 & 25 & 25 & 15 & 15 & 10 & 0 & 0 & 0 \\ a_2 & 30 & 105 & 60 & 40 & 20 & 15 & 15 & 0 & 0 & 0 \\ a_3 & 25 & 60 & 40 & 30 & 15 & 15 & 10 & 0 & 0 & 0 \\ a_7 & 25 & 40 & 30 & 65 & 25 & 15 & 10 & 0 & 0 & 0 \\ a_{10} & 15 & 20 & 15 & 25 & 70 & 10 & 10 & 0 & 0 & 0 \\ a_8 & 15 & 15 & 15 & 15 & 10 & 60 & 10 & 18 & 18 & 18 \\ a_1 & 10 & 15 & 10 & 10 & 10 & 10 & 50 & 18 & 18 & 18 \\ \hline a_9 & 0 & 0 & 0 & 0 & 0 & 18 & 18 & 40 & 40 & 40 \\ a_4 & 0 & 0 & 0 & 0 & 0 & 18 & 18 & 40 & 40 & 40 \\ a_5 & 0 & 0 & 0 & 0 & 0 & 18 & 18 & 40 & 40 & 40 \end{array} \right\}$$

Fig. 2.5 Clustered Affinity Matrix

Partitioning: From CUM, 2 non-overlapping fragments are identified. It selects one point on the main diagonal of CUM such that the value of Z is maximized:

$$Z = (QU*QL) - QI^2$$

The point divides the CAM into two partitions: upper block U and lower block L. Each block specifies a fragment. QU and QI represent number of queries that need to access only properties in upper block and lower block respectively. QI is the number of queries that need to access properties from both fragments.

b) Pivoting/Unpivoting Phase: In case there is a change in the query workload of the system, a readjustment in the table schema is required. For this, pivot and unpivot operations are used. The pivot operation is used to move the data from triple table to the property tables, and the unpivot operation to move back the data from property tables to the triple store. A combination of these two operations are used to move properties between property tables.

2.2.2.2 Workload Aware Hybrid Partitioning WAHP

This approach uses a combination of both horizontal and vertical partitioning based on the query workload [7][8]. The method is carried out in following phases:

a) WAHP-1: Hot schema is identified in this phase which leads to the horizontal partitioning of main dataset. From the given workload, frequency of access of each property is identified. Properties with frequency above than a threshold form a hot schema. Frequency threshold is set in the experiment as the 80% of mean frequency of all properties.

b) WAHP-2: After the hot schema is identified, we map the relevant workload to that schema. It creates temporary tables and then store corresponding records in the workload aware clusters. These are the vertically partitioned tables of the original tables. The data populated in these clusters is the horizontal partition of the relevant workload.

In the experiments conducted, using this approach, 9% data in workload aware clusters can answer 73% of the hottest query workload. It gives an average execution time gain of 37% than the original dataset.

2.2.2.3 Identification of Hot and Cold Data

This approach exploits the skewedness of the access pattern of data in an OLTP system [9]. It identifies the hot data which is frequently accessed and separates it from the rest of data which is either randomly accessed or not accessed at all. Hot data is identified using log of record access in database. When the workload is run, a separate log can be maintained for their record accesses having record ids and time stamp. A fast algorithm using exponential smoothing is executed to accurately classify data as hot or cold using the log file. Here we have an option of either inline or offline analysis. If we use an inline approach then each records access frequency estimation is maintained in main memory and updated on every record access. In an offline estimation approach, there is less overhead as the log can be operated on a separate machine to find record access estimate which gives a flexibility in where, when and how to analyse the log. The method is discussed in detail in Chapter 3.

2.3 Partitioning of RDF data for Mixed Workload

To partition dataset based upon the mixed workloads i.e. Online Transactional Processing OLTP and Online Analytical Processing OLAP workloads, a Relevance Based Partitioning RBP approach is designed [10]. It identifies lists of most recently accessed tuples and columns which are hot for transactional and analytical workload respectively. To reflect both analytical and transactional workload patterns, it horizontal partitions the table. But instead of partitioning the whole table, it creates tailored partition of each column of the table. It moves cold data to a cheaper Storage Class Memory SCM so that it does not occupy costlier main memory.

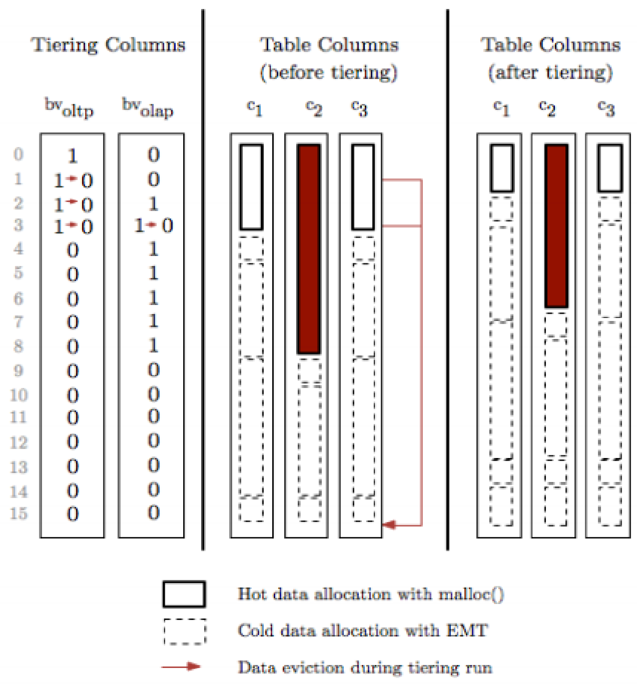

Tiering Columns
bv_{oltp}
bv_{olap}
Table Columns (before tiering)
c_1 c_2 c_3
Table Columns (after tiering)
c_1 c_2 c_3
Hot data allocation with malloc()
Cold data allocation with EMT
Data eviction during tiering run

Fig. 2.6 Data Tiering for Mixed Workload

Hot Data Views HDVs are identified from the table after workload execution. These define hot data within a table. It separates the hot and cold data and also checks whether a query can be answered using only hot partitions. A number of HDVs can be created for a table. After identification of HDVs a tiering run is introduced during which new tuples are considered hot and tuples in a hot partition that are not relevant are moved into a new cold partition of the column. Each column is partitioned into a single hot partition and multiple cold partitions. Hot data views of each column are kept in main memory whereas cold partitions are kept on secondary storage.

Chapter 3

Log Based Method LBM

This chapter discusses the partitioning technique used in the thesis. It is based upon the type of query workload executed on the IoT dataset. The method used in the thesis, Log Based Method LBM is presented and the various steps involved to execute it are also discussed.

LBM is used to identify frequently accessed records in the IoT dataset. It exploits the skewedness in record access pattern by analysing log of record accesses. The log file is analysed to find the access frequency of each record. As there are skewed access patterns in OLTP systems, not all records have same access frequency. It varies based upon the output of each query and their workload. Records with high frequencies are considered as hot records. Rest all records are considered as cold records. Of all records, hot records are queried frequently whereas cold records are accessed infrequently or not at all. Algorithms implementing LBM are run frequently to accurately classify data. These are based on exponential smoothing.

3.1 Exponential Smoothing ES

Smoothing is a statistical method used to identify demand patterns based upon the previous values. It basically removes random variations from the historical patterns and smooth it out to estimate future demands. Exponential Smoothing is one such smoothing method in which weights are applied to previous values. More weight is given to recent period and lesser weights are given to the earlier periods. Smoothing factor determines the weight to be given to each previous period. If we use 20% as the smoothing factor, the weight given to most recent period's demand will be 20%, the weight of the next most recent period will be 80% of 20% i.e. 0.16%, weight of the period before that will be 80% of 80% of 20% i.e. 0.128% and so on.

- **Method:** As each record's entries are accessed in the log file, exponential smoothing is applied to find its estimated access frequency as follows:

$$est_r(t_{curr}) = \alpha * x_{t_{curr}} + (1-\alpha) * est_r(t_{curr-1})$$

where, r = record which is accessed in the log file

t_{curr} = start time of current time slice

$est_r (t_{curr})$ = estimated access frequency of the record r at current time slice.

$est_r (t_{curr-1})$ = estimated access frequency of the record r at previous time slice.

x_{tcurr} = value of actual frequency at current time slice.

α = delay factor

Current estimation is calculated based upon its current as well as previous access. If a record is accessed in the current time slice, we keep x_{tcurr} as 1 and 0 otherwise.

- **Effect of $\boldsymbol{\alpha}$:** It is the delay factor which determines the weight given to each estimate. A smoothing factor of 20% implies α as 0.2. Value of α can vary in interval [0,1]. Consider a record r is observed in time slices <1,2>, <3,4> and <7,8>. According to exponential smoothing, frequency estimation of record r is calculated as shown in Fig. 3.1:

$$est_r(1) = 1$$
$$est_r(2) = \alpha*0 + (1-\alpha) * est_r(1) = (1-\alpha) * est_r(1)$$
$$est_r(3) = \alpha*1+(1-\alpha) * est_r(2) = \alpha + (1-\alpha) * (1-\alpha) * est_r(1)$$
$$est_r(4) = \alpha*0 + (1-\alpha) * est_r(3) = (1-\alpha) * est_r(3) = (1-\alpha) * \alpha + (1-\alpha) * (1-\alpha) * (1-\alpha) * est_r(1)$$
$$est_r(5) = \alpha*1+(1-\alpha) * est_r(5) = \alpha + (1-\alpha) * (1-\alpha) * \alpha + (1-\alpha) * (1-\alpha) * (1-\alpha) * (1-\alpha) * est_r(1)$$

Fig. 3.1 Effect of α

The value of $est_r(5)$ has more dependence on the value of $est_r(4)$ and lesser on the previous values such as $est_r(3)$, $est_r(2)$ and $est_r(1)$ in decreasing order. It determines the weight given to new observations and how quickly to decay old estimates. In the log file, the value of   determines how much importance is to be given to the older values as compared to newer values of a record's access. It gives the different importance of recent and earlier access of log entries. So, it creates aging in log file entries.

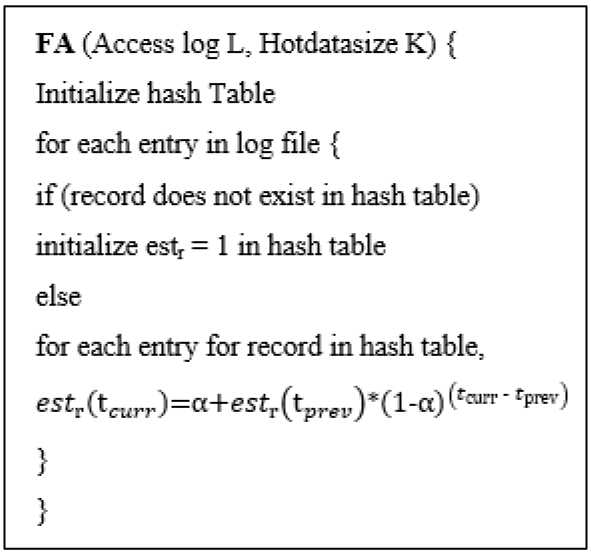
Following two algorithms based on ES can be used to analyse log file:

3.1.1 Forward Algorithm FA

The log is scanned from beginning time slice ($t_b = 0$). An estimated access frequency is calculated for each record accessed in the log file. Upon encounter of a record in log file, its estimated access frequency is updated as follows:

$$est_r(t_{curr}) = \alpha + est_r(t_{prev}) * (1-\alpha)^{(t_{curr}-t_{prev})}$$

Here, estimated access frequency for a record is calculated only one it is accessed on that time slice. Thus, t_{prev} is time slice when record r was last observed. To avoid updating the estimate of every record at every time slice, the previous estimated are decayed using factor $(1-\alpha)^{(t_{curr}-t_{prev})}$. The exponent ($t_{curr}$ - t_{prev}) allows the estimate to catch up by decaying the previous estimate across time slices when r was not observed in the log. When the algorithm finishes scanning the complete log file, we get the estimated access frequency of each record. We now find the top K% records with highest estimated access frequencies. These top K% ranked records are considered as the hot records.



```
FA (Access log L, Hotdatasize K) {
Initialize hash Table
for each entry in log file {
if (record does not exist in hash table)
initialize $est_r$ = 1 in hash table
else
for each entry for record in hash table,
$est_r(t_{curr})=\alpha+est_r(t_{prev})*(1-\alpha)^{(t_{curr}-t_{prev})}$
}
}
```

Fig. 3.2 FA Pseudo Code

- **FA Analysis:** The total number of entries in the log file is L. If record r_1 is accessed k_1 number of times, r_2 is accessed k_2 number of time, r_n is accessed k_n number of times,

$$k_1 + k_2 + k_3 + \ldots + k_n = L$$

Total number of accesses for record r_1 = $k_1 * (k_1 - 1) / 2$

Total number of accesses for record r_2 = $k_2 * (k_2-1) / 2$

Similarly, total number of accesses for record r_n = $k_n * (k_n-1) / 2$

Total number of accesses ~ $k_1^2 + k_2^2 + k_3^2 + \ldots + k_n^2$

So, time complexity for forward algorithm is $(k_1^2 + k_2^2 + k_3^2 + \ldots + k_n^2)$. Space complexity is L as for each record the space required is the number of times it is accessed.

3.1.2 Backward Algorithm BA

The previous algorithm requires scanning of complete log file. It also requires space to store estimated access frequencies for each unique record in log file. Using the following algorithm, the time and space requirement for record access estimation can be reduced. The algorithm attempts to pre-empt early. It involves scanning the log in reverse direction and finding access frequency estimates for each record. It also tries to store only the records which are in contention for hot records and reject all others thus its memory requirement is also proportional to the number of hot records instead of all records.

```
BA (AccessLog L, HotDataSize K) {
    Hash Table H ← initialize hash table      /* Initialisation Phase start */
    Read back in L to fill H with K unique records with calculated bounds
    kthLower ← RecStats r ∈ H with smallest r.loEst value
    acceptThresh ← $\lfloor t_e - \log(1-\alpha)\text{kthLower} \rfloor$      /* Initialisation Phase end */
    while not at beginning of L do{      /* Calculation  Phase begins */
        rid ← read next record id from L in reverse
        RecStats r ← H.get(rid)
        if r is null then      /* disregard new record ids read after acceptThresh time slice */
            if L.curTime ≤ acceptThresh then goto line 6
            else initialize new r
        end if
        update r.estb using Equation 3
        H.put(rid,r)   /* Calculation Phase ends */
        if end of time slice reached then      /* filter step starts - inactivate all records that cannot be in hot set*/
            ∀r ∈ H update r.upEst and r.loEst using Equations 4 and 5
            kthLower ← find value of kth lower bound value in H
            ∀r ∈ H with r.upEst ≤ kthLower, remove r from H
            if num records ∈ H is K then goto line 25
            acceptThresh ← $\lfloor te - \log_{(1-\alpha)} \text{kthLower} \rfloor$
        end if      /* filter step ends */
    end while
    return record ids in H with r.active = true
}
```

Fig. 3.3 BA Pseudo Code

The algorithm given in Fig. 3.3 proceeds in following phases:

i. **Initialisation phase:** the algorithm begins with analysing log file from end. A hash table is maintained to store hot records. As and when a record is accessed, its backward access frequency estimate is calculated as:

$$estb_r(t_{curr}) = \alpha * (1-\alpha)^{(t_e - t_{curr})} + estb_r(t_{last})$$

where, t_e= end time slice i.e. start time of last record access.

t_{last}= time slice when the record was last encountered.

t_{last} is less than t_e and t_{last} is greater than t_{curr}, as we are accessing log in reverse order. For each access of record, its upper and lower bound access frequency estimate is also calculated for that time. They signify the highest and lower value access frequency estimation can have for that record. Upper bound for the record at current time slice can be calculated as:

$$upEstb_r(t_{curr}) = estb_r(t_{curr}) + (1-\alpha)^{(t_e - t_{curr}+1)}$$

It is calculated by assuming that the record has been accessed every time we are scanning backward in the log file. So this is the highest value the backward access frequency estimate can have for the record at this time. Similarly, the lower bound is calculated as:

$$loEstb_r(t_{curr}) = estb_r(t_{curr}) + (1-\alpha)^{(t_e - t_b+1)}$$

where, t_b= start time slice i.e. start time of first record access.

Lower bound is calculated by assuming that the record will never be encountered again while moving backwards in the log file. So, it is the lowest value the backward access frequency estimate can have for the record at this time. The log is read backwards till the point we have encountered K unique records. Backward access frequency estimate, upper bound and lower bound is calculated and stored for these K records. From these records, we find the Kth lower bound value. Any record having upper bound less than this value is rejected because the top K records will have estimation value greater than or equal to the Kth lower bound value. Thus, any record with upper bound less than this value cannot have estimation value greater than those K records, and hence they do not qualify to be in top K hot records.

The Kth lower bound value is translated to a time slice in the log file named as accept threshold as:

$$Accept\ Threshold = t_e - \lfloor \log_{(1-\alpha)} K^{th}\ lower\ bound \rfloor$$

It represents a time slice in log at and beyond which we can discard any record accessed as they will never have upper bound greater than the Kth lower bound. So those records cannot be in contention for hot records. Thus, we do not store estimation values for each record and it limits the memory requirement of the algorithm.

ii. **Calculation Phase:** the log file is further accessed reading next record access. If the record has never been accessed before, we compare if its current time slice is greater than the accept threshold. If so then the record is stored and its backward record estimation is calculated. If not then the record is discarded and not stored as it cannot have upper bound of estimation value greater than the K^{th} lower bound. If the record accessed at current time slice has already been accessed and is stored, then the value of its backward access frequency estimation is updated.

iii. **Filter Phase:** the phase is executed at the end of each time slice. Upper and lower bounds for each record stored is updated. From the lower bounds, K^{th} lower bound value is calculated. As before, the records with upper bound less than this value are removed from stored records. So, we have only those records which are in contention for hot records. If number of records that are stored is K, then it is the set of hot records that is required. We terminate the algorithm at this point. If not, then the accept threshold is recalculated based on the new K^{th} lower bound value. As the new K^{th} is greater than its earlier value, the new accept threshold is greater than the earlier one. This moves the accept threshold closer to the current scan point in the log file. The log is further read backwards. Removing records from the hash table ensures the less memory requirement of the algorithm.

As shown in Fig. 3.4, consider the case when K=3. At time t_n, R_3 represents the K^{th} lower bound value. Record R_1 is discarded as its upper bound of estimation value is even less than the K^{th} lower value. Records R_2, R_3, R_4, R_5 and R_6 are in contention for hot set.

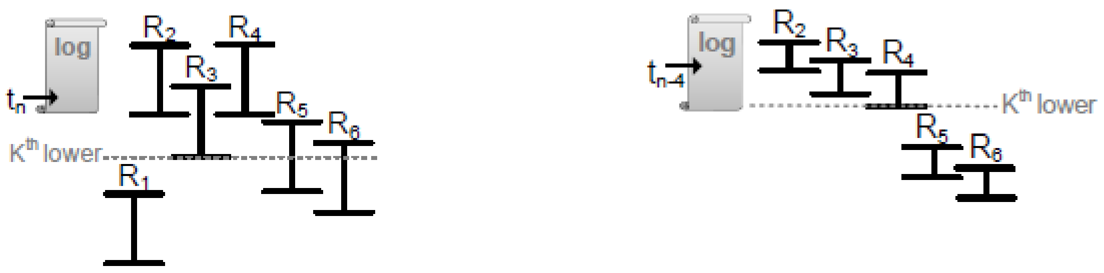


Fig. 3.4 Records in Contention for Hot Set

At time slice t_{n-4}, Kth lower bound value is represented by R_4. Records R_5 and R_6 are rejected as they have upper bound values less than Kth lower bound value. The records in contention for hot records are R_2, R_3 and R_4. As the number of records are equal to K, thus the algorithm is terminated. A record is updated only once in a time slice. Any further access of the record in same time slice is ignored. If the algorithm is not pre-empted early, then it reads the log file till the end. At that point, the upper and lower bound values for each record is same. We rank each record according to value of their backward access frequency estimate. Top K records are found and are considered as hot records. Rest all records are cold records.

- **BA Analysis:** The total number of entries in log file is L. K is the number of records that can be considered as hot records. For BA, the time complexity is $\Theta(L*K)$. The space complexity is $\Theta(K)$ as we store only the records which are in contention for the hot set.

As it is expected to speed up the execution of FA and BA, implementation of these algorithms can also be carried out in parallel using multiple threads.

3.2 Implementation of LBM

LBM is implemented using the above-mentioned algorithms. It involves creating a log file by query execution, then sampling the log file to reduce the size of log file followed by its analysis using FA or BA to find hot records. It is carried out in steps as given in Fig. 3.5:

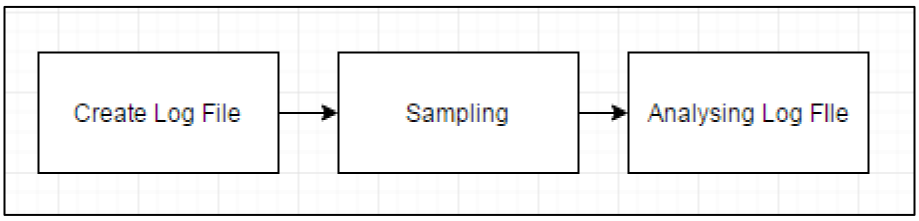


Fig. 3.5 LBM Flow

3.2.1 Creating Log File

For an application, the queries and their pattern of occurrence is called its query workload. Output of queries are logged in their order of execution in a separate log file. A query can access more than one record. Each record access is logged as

<record ID, [t_n,t_{n+1}]> where record ID is unique identifier for each record and [t_n,t_{n+1}] is the time slice of the record access . t_n is the start time of record access and t_{n+1} is the end time of its access.

3.2.2 Sampling

Depending upon the system load, the log file can be very huge. If we log every access of records we get most accurate estimates. But in case of millions of entries in log file, it is difficult to analyse it. Therefore, we can consider analysing only some parts of the log file. The contents of log file can be reduced by sampling.

In our thesis, sampling can be done in two ways. First, logging only few record accesses at the time of query execution, so only a sample of accesses are recorded in the log file. Second, to log each record access and later at the time of analysis applying sampling to select some log accesses. The sampling can be done by using any systematic sampling technique. It means that the samples are drawn according to a random starting point and a fixed periodic interval. This interval depends upon the size of log file that we need to process. At the end of this step, we have a sampled log file which is smaller in size than the original log file.

3.2.3 Analysing Log File

The log is analysed to find the access frequency of each record. It can be done using implementation of the any of the algorithms, FA or BA. The log file can be analysed offline or online. If it is analysed online, it will require the resources of system running transaction workload. So, it is a better approach to analyse it offline. It also gives flexibility in how, when and where to analyse the log. It can be done on a separate machine so that it does not interrupt system's normal operation.

- **Value of K:** Hot records are the top K% ranked records with highest record access frequency. The value of K depends upon the available main memory in the system. It is generally kept below 10% as the rest of the memory is required for other CPU processes like logging, locking, latching, buffer management. So, we can safely move data of size worth 10% of RAM size into main memory for the efficient working of the system.

At the end of this step, we have the records which can be considered as hot records. We partition the triple table by separating these records from cold records which gives hot data and cold data table.

Chapter 4

LBM Experiments

The chapter describes the resources used in the experiments to partition IoT data. It explains the steps followed for the implementation of LBM. It also explains how to evaluate the performance gain of the system.

4.1 Experimental Flow

The experiments are carried out using an IoT dataset and its query set. IoT data used is static and is represented in relational format. The workload is also static with known queries and their ranking.

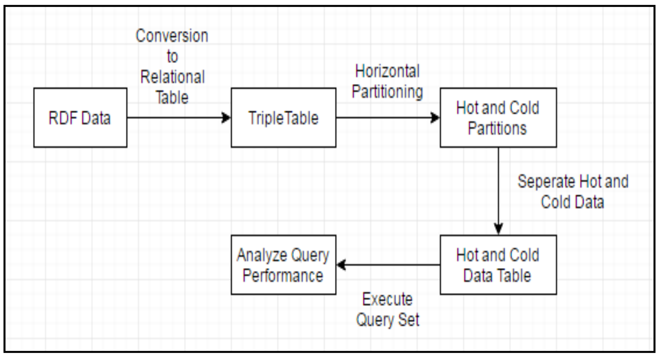


Fig. 4.1 Experimental Flow

The experimental flow is as shown in Fig.4.1. The dataset is converted and stored in a triple table. The log files are generated to log the record accesses as the queries are executed. The data access pattern is identified using LBM and the triple table is partitioned in hot and cold sections. The query set is executed on both triple table and the hot and cold sections to analyse the system performance.

4.2 Experimental Setup

The LBM experiment is performed using an IoT dataset. The dataset is static and queries are known beforehand with their frequencies.

4.2.1 Data Set

The experiment uses benchmark RDF datasets – Linked Sensor Data (LSD) and Linked Observation Data (LOD). These are the weather data aggregated by Department of Meteorology at the University of Utah since 2002.

- LSD: It is a RDF dataset containing description of some weather stations in US. It contains location attributes of the sensors located at each weather station. These attributes are latitude, longitude, elevation, link to location in Geonames near to weather station, distance from Geonames location to weather station. These sensors at each weather station measure temperature, pressure, wind speed, visibility and many other phenomena.

- LOD: It is a description of hurricane and blizzard observations collected by above mentioned sensors. Apart from the measurements of these phenomenon, it also contains their unit of measurement and time instant when these are taken. It contains information regarding several hurricanes and blizzards active in US in 2002. The original dataset contains more than a billion triples and takes several GBs of storage space.

Table 4.1 Data Set Specifications

Size on disk	24GB
Number of triples	114343293
Number of unique properties	19

- Properties: The dataset we have used contains 19 unique properties describing characteristics of 9,702 sensors. Following are the properties defined in the dataset:

Table 4.2 Property Values in LOD-LSD Dataset

S.No.	Property
1	"<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#processLocation>"
2	"<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#distance>"
3	"<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>"
4	"<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#floatValue>"
5	"<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#hasLocation>"
6	"<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#hasLocatedNearRel>"

7	"<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#observedProperty>"
8	"<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#uom>"
9	"<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#parameter>"
10	"<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#samplingTime>"
11	"<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#result>"
12	"<http://www.w3.org/2006/time#inXSDDateTime>"
13	"<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#generatedObservation>"
14	"<http://www.w3.org/2003/01/geo/wgs84_pos#long>"
15	"<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#hasSourceURI>"
16	"<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#procedure>"
17	"<http://www.w3.org/2003/01/geo/wgs84_pos#alt>"
18	"<http://www.w3.org/2003/01/geo/wgs84_pos#lat>"
19	"<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#ID>"

- Conversion of Dataset into Relational Format: The LOD and LSD datasets are available in XML format. They are converted using Apache Jena Parser into RDF triples. These triples are then parsed and stored in the relational format. Each triple is stored as a separate row in a table with 3-column schema: subject, property and object. An additional information is used to identify each record in the table as Object ID (OID). The memory required to store this non-RDF and access information is not significant so we can use it as row identifier. The values of subject and property columns are URIs stored as strings. The value of object column is either an URI or a string literal. For our experiment, we have used PostgreSQL database v9.5 on pgAdmin development platform v1.22.

4.2.2 Query Set

We have identified a set of 28 queries. We have tried to create a realistic workload by assigning appropriate frequency to each query. Frequent and most frequent queries are identified. Queries are arranged in non-increasing order of their frequencies. From this,

- top 50% queries are considered as frequent queries. Out of 28 queries, 11 are frequent queries.
- top 50% frequent queries are considered as most frequent queries. Out of 28 queries, 6 are most frequent queries.

Queries are written in SQL and are run on pgAdmin platform.

4.2.3 Hardware and Software Configuration

The database is stored on a 2GB RAM machine with Windows7 as Operating System OS using PostgreSQL 9.5.3. The queries are executed and log is generated on this system. The log file is then analysed using LBM on a system with 32GB of RAM and Scientific Linux release 7.1 OS. The analysis is done using java 1.7 on eclipse 4.6.0 IDE.

4.3 Implementation of LBM

Implementation of the method involves generation of log file and its sampling followed by its analysis using Forward or Backward Algorithms.

4.3.1 Log Generation

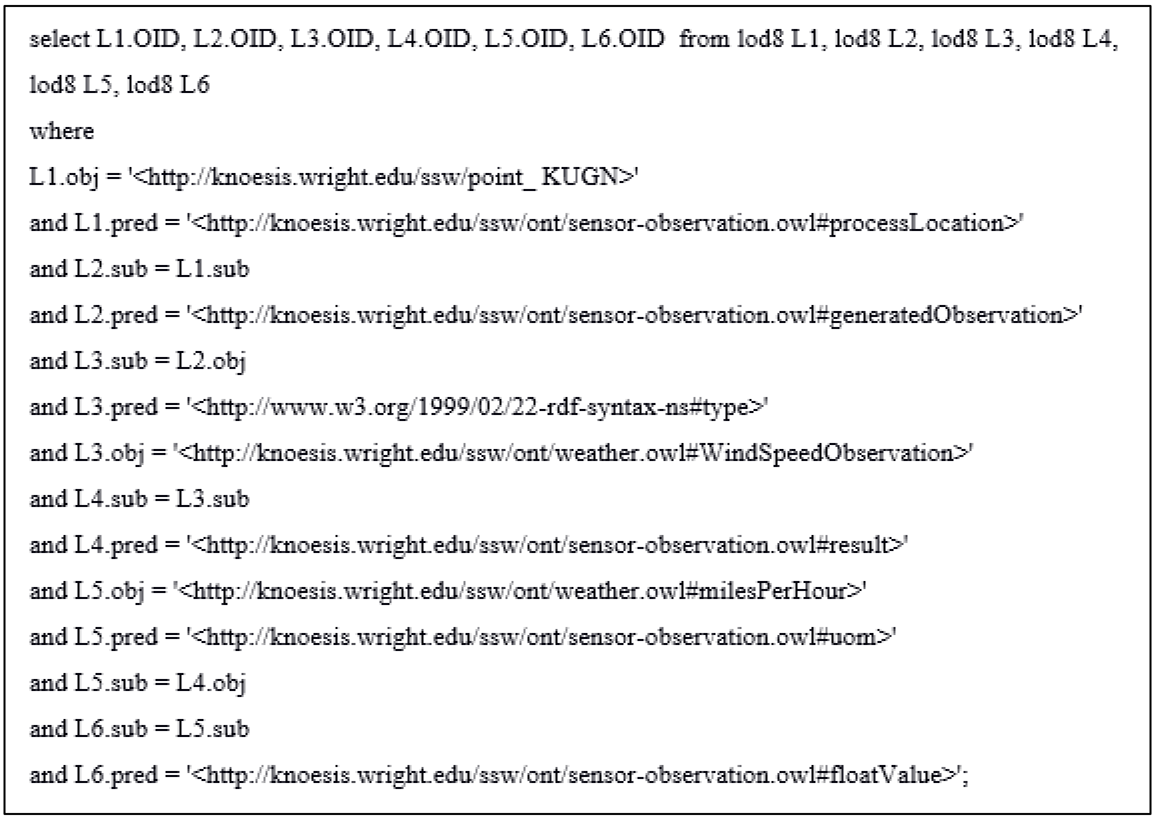
To generate the log file, queries are executed in a fashion defined beforehand the experiment.

- What is logged: As the data is in RDF format, all the properties for a resource are not stored in a single row in table. For a subject, each property is stored as a separate row. To solve a typical query, we must consider multiple rows. Thus, all the rows are required to be logged in the file as those records are also required to give correct answer to the query. e.g. to determine wind observation at location "<http://knoesis.wright.edu/ssw/point_A21>", sql query is as shown in Fig. 4.2:

```
select L6.obj from lod8 L1, lod8 L2, lod8 L3, lod8 L4, lod8 L5, lod8 L6
where L1.obj = '<http://knoesis.wright.edu/ssw/point_ KUGN>'
and L1.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#processLocation>'
and L2.sub = L1.sub
and L2.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#generatedObservation>'
and L3.sub = L2.obj
and L3.pred = '<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>'
and L3.obj = '<http://knoesis.wright.edu/ssw/ont/weather.owl#WindSpeedObservation>'
and L4.sub = L3.sub
and L4.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#result>'
and L5.obj = '<http://knoesis.wright.edu/ssw/ont/weather.owl#milesPerHour>'
and L5.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#uom>'
and L5.sub = L4.obj
and L6.sub = L5.sub
and L6.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#floatValue>';
```

Fig. 4.2 Actual SQL Query

The result contains 304 wind observation values. But the query can only be solved if we log all the triples required to solve this query. So, the query run to log each such record is as shown in Fig. 4.3:



```
select L1.OID, L2.OID, L3.OID, L4.OID, L5.OID, L6.OID  from lod8 L1, lod8 L2, lod8 L3, lod8 L4,
lod8 L5, lod8 L6
where
L1.obj = '<http://knoesis.wright.edu/ssw/point_ KUGN>'
and L1.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#processLocation>'
and L2.sub = L1.sub
and L2.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#generatedObservation>'
and L3.sub = L2.obj
and L3.pred = '<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>'
and L3.obj = '<http://knoesis.wright.edu/ssw/ont/weather.owl#WindSpeedObservation>'
and L4.sub = L3.sub
and L4.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#result>'
and L5.obj = '<http://knoesis.wright.edu/ssw/ont/weather.owl#milesPerHour>'
and L5.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#uom>'
and L5.sub = L4.obj
and L6.sub = L5.sub
and L6.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#floatValue>';
```

Fig. 4.3 SQL Query Executed to Create Log File

Each record access is logged as <Record ID, Timestamp>. Here record ID is the OID which is unique identification number for each record and timestamp is (t_n,t_{n+1}). t_n is the start time of this record access and t_{n+1} is its end time. The log is maintained in a log table stored in the same database as our triple table.

The different query execution patterns used are as follows:

- Random order with fixed query frequency: Each query is assigned a frequency which signifies the number of times a query will execute. For order of execution, a query is selected randomly and it is executed on the dataset. It runs the number of times its frequency. Its records are logged in order of their access.

- Random order with fixed total queries frequency: A total number of query runs are fixed. Frequency of each query is not known beforehand. The queries are run in a random order.

- Fixed order with fixed query frequency: Each query is assigned a fixed frequency. A fixed order of query execution is also decided. At their turn, each query runs the number of times its frequency.

- Zipf order: To efficiently represent the practical workload, a log file is generated using zipf distribution. We have identified an order of queries based upon their likelihood of querying. According to zipf distribution, for each query i,

$$R_i * f_i = c$$

  where, R_i is the rank of query i, f_i is the frequency of execution of query i and c is a constant. By fixing a value of constant, we can determine frequency of each query as we know its rank. To generate the log file, the queries are executed in non-increasing order of their frequencies. The most frequent query is followed by the second frequent query and so on. The least frequent query is executed at the end. Each query logs the record accesses as <Record ID, Timeslice> in a log file. A log of any size can be generated by changing the value of constant. A value of constant as 100.0 generates a log of 47.5M record accesses. As we have created a zipf distribution of queries so their accessed records will also follow the same distribution.

4.3.2 Log Analysis

From the log table, the entries which are recorded within the observation time are drawn out in a file. This file is then analysed to estimate the records access frequencies using FA and BA. To compare results obtained by these methods, we also calculate the actual frequency of occurrence of each record using the log file. The analysis is done on a machine with 32GB of RAM and Scientific Linux release 7.1 OS using java 1.7 on eclipse 4.6.0 IDE.

4.3.2.1 Calculation of Actual Access Frequency

The log file is analysed to find the actual access frequency for each record. It is done in to compare the results of FA and BA. For the ideal case, the hot records identified by the actual analysis of complete log file should be the same as those identified by FA and BA.

4.3.2.2 Setting Parameters for Forward Algorithm

The entries of the log file are taken in order of their occurrence. The value of α is decided beforehand from [0.01-0.05]. A hashmap is maintained to store estimated frequencies for each record. When a record ID is encountered for the first time, an entry is made in the hashmap to store its estimated frequency as 1 and its time of occurrence. On the subsequent accesses of a record r at time t_{curr}, it updates the estimated frequencies based upon its access at previous time slice t_{prev} using:

$$est_r(t_{curr}) = \alpha + est_r(t_{prev}) * (1 - \alpha)^{(t_{curr} - t_{prev})}$$

If the record is accessed more than once in same time slice, its first access is considered and rest all accesses are ignored. At the end of the log file, we find the top K% ranked records with highest estimated access frequencies which are considered as the hot records.

4.3.2.3 Setting Parameters for Backward Algorithm

The entries of the log file are considered from end of the log file i.e. the record accesses encountered at end of observation period will be read first. When a record is encountered for the first time, its access frequency is assigned as . On subsequent access of record r at time slice t_{curr}, the access frequency is estimated based upon its access at time slice t_{last} using:

$$estb_r(t_{curr}) = \alpha * (1-\alpha)^{(t_e - t_{curr})} + estb_r(t_{last})$$

Upper and lower bound values of access frequency is also estimated. If a record is accessed more than once in the same timeslice its subsequent accesses are ignored. The log file is read backwards till K unique records are encountered. The least value of lower bound estimated frequency is found from these K records and is translated to a time slice in the log file named as accept threshold:

$$Accept\ Threshold = t_e - \left\lfloor \log_{(1-\alpha)} K^{th}\ lower\ bound \right\rfloor$$

It restricts the memory requirement of the system as any record accessed beyond this time slice cannot be in contention for hot record and thus the log is not read beyond this time slice. The log is further read and at the end of each time slice, the upper and lower bound values of estimated frequencies are updated from which Kth lower bound value is calculated. As before, the records with upper bound less than this value are removed from stored records. So, we have only those records which are in contention for hot records. If number of records that are stored is K, then algorithm is terminated. The remaining records in the hashtable are the hot records.

4.3.2.4 Loss in Hit Rate

Hit rate is calculated to determine the difference between actual hot records and estimated hot records. Actual hot records are the records identified as hot by actual record access frequency calculation. Estimated hot records are those identified as hot by either FA or BA. As a record can be accessed more than once, a record miscalculated as cold will cause a loss in hit rate when the respective query will be executed. So, to determine hit rate we calculate the total number of times we are going to experience a loss for each miscalculated record. Consider the following case:

k = actual number of hot records (b_1,b_2,…..,b_k).

Actual access frequency of $b_1 = g_1$

Actual access frequency of $b_2 = g_2$

Similarly, actual access frequency of $b_k = g_k$

p= number of miscalculated records as cold (a_1,a_2,….,a_p)

Actual access frequency of $a_1 = f_1$

Actual access frequency of $a_2 = f_2$

Similarly, actual access frequency of $a_p = f_p$

We will not find records a_1,a_2,….,a_p in hot data section. Every time when these records are required, we will experience a loss in hit rate as:

$$\text{Total loss in hit rate} = \frac{\sum_{i=1}^{k} g_i - \sum_{i=1}^{p} f_i}{\sum_{i=1}^{k} g_i} * 100$$

4.3.3 Partitioning Data into Hot and Cold Data Sections

The data considered as hot by FA or BA is moved in a separate table called hot data table. These records are identified from the triple table and are removed from it, the resultant table is renamed as cold data table. Thus, we have 2 tables: hot data table containing hot data triples; and cold data table containing remaining triples from triple table which are not accessed frequently and are considered as cold data.

4.3.4 Experiments on Partitioned Data

To find out the performance of system after partitioning and to find the efficiency of algorithm, following experiments are performed:

- **Query Performance Analysis:** It is done in order to compare the query performance in triple table and partitioned data. The queries of IoT system are run on the partitioned data sections. Some queries are answered using only hot data section. Some queries give partial results when run on hot data section. To get the complete results we execute it on both hot and cold data sections and combine their results. This gives the complete result of the query. The time taken to get the complete result set of each query is calculated using 3 hot runs. Hot run is when the data required to answer a query is available in the Postgres cache. This query execution time is compared with the query execution time required in case of triple table. We determine how many queries, frequent queries and most frequent queries are fully answered by only hot data section.

- **Value of K:** The different value of K which determines the amount of hot data and the amount of workload it can answer is experimentally determined.

- **Log Scaling:** The efficiency of algorithm is experimentally checked to find out by incrementally scaling the amount of log file in order to find out the hot data.

Chapter 5

Results and Discussions

This chapter contains the results obtained from experiments to implement LBM. The analysis of query performance in terms of query execution time is presented along with gain experienced in case of partitioned dataset. The analysis of the algorithms is done based on various parameters like size of log file and value of K. It discusses the effects of data and log file scaling on the algorithm performance.

The resources used for the experiment are as follows:

- Data Stored at: 2GB RAM, Windows 7 OS System
- Dataset Size in Relational Table: 2486MB
- No. of Rows in Table: 1,05,05,690

5.1 Hot Data and Workload Answered

The value of K determines the data that can be considered as hot from the dataset. Hot data contains the IoT triples that are considered most frequently and they alone can answer most of the queries posed on the system. It depends upon the value of K that how much workload can be answered from hot data alone. The value of K is determined on the amount of main memory available in the system. As we have 2GB RAM machine, we can keep use 10% i.e. 200MB of space to store hot data. As the size of triple table is 2.5 GB, we can store 8% of its data i.e. 199MB of data as hot data set as shown in Table 5.1. So, the value of K in case of this experiment is fixed as 8%.

Table 5.1 Hot Data Size with Change in K

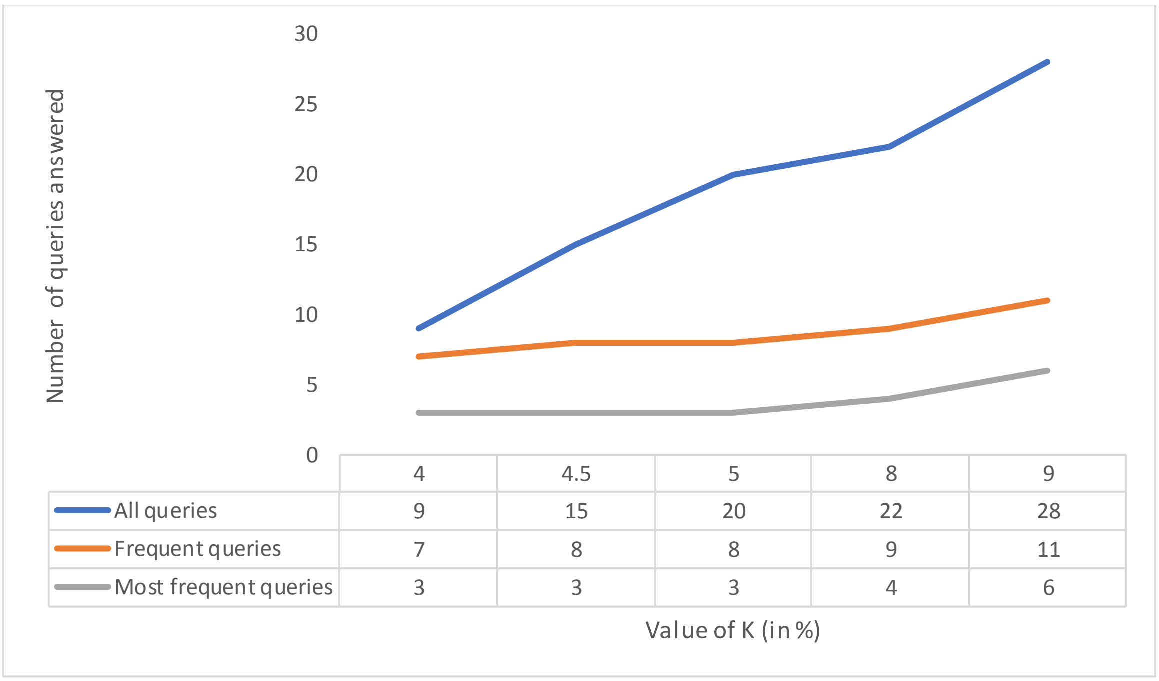
K (%)	No. of rows in hot data table	Size of hot data table in memory
4	4,26,331	106
4.5	5,19,327	129
5	5,87,200	144
8	8,91,193	199
9	9,12,487	218

- Query Workload answered QW$_{ans}$: Each query has a frequency of occurrence. Total workload of the system is the summation of frequencies of all queries targeted for the system. Percentage of amount of workload answered is given by:

$$QW_{ans} = \frac{\sum_{i=1}^{n}(M * f_i)}{\sum_{i=1}^{n}(N * f_i)} * 100$$

  Where, M is the queries answered by the hot data and N is the total number of queries.

We have found the following observation displaying the effect of change in value of K over the queries answered:



	4	4.5	5	8	9
▬ All queries	9	15	20	22	28
▬ Frequent queries	7	8	8	9	11
▬ Most frequent queries	3	3	3	4	6

Fig. 5.1 Workload Answered with Change in K

Using 8% as hot data, we can answer 78.57% of all queries and 86.4% of workload as shown in Fig.:

Table 5.2 Query Workload Answered

Query specification	% of queries answered	% of workload answered
All queries	78.57	86.4
Frequent queries	81	77.56
Most frequent queries	66.67	68

5.2 Query Execution Time QET

The queries are executed on partitioned tables and the QET for each query is compared with QET taken by triple table. QET in both cases is measured as average of time taken for 3 hot runs. The average QET per query is found to be less after partitioning as shown in Table 5.3:

Table 5.3 QET in Partitioned Table

Query	Time taken per query in Main Table (min)	Time taken per query in Partitioned Table (min)	Time Gain (min)
All queries	10.36	6.99	3.46
Frequent queries	14.097	3.19	10.907
Most Frequent Queries	12.072	1.68	10.392

There is a significant decrease in QET for frequent and most frequent queries. This makes our system processing queries faster most of the times.

5.3 Time Gain TG

For each query i, time gain is calculated as:

$$TG_i = (QET_i)_{\text{triple table}} - (QET_i)_{\text{partitioned table}}$$

In terms of percentage, it is calculated as:

$$TG_i(\%) = \frac{(QET_i)_{triple\,table} - (QET_i)_{partitioned\ table}}{(QET_i)_{triple\,table}} * 100$$

In our experiment, the TG(in %) attained in case of partitioned table is as shown in Table 5.4:

Table 5.4 Time Gain in Partitioned Table

Query specification	Time gain (%)
All queries	67.53
Frequent queries	69.83
Most frequent queries	86.08

Average QET of hot run for triple table is 10.36 min. After the partition, average QET of hot run is reduced to 6.99 min. Thus, we are getting a time gain of 67.53%. For frequent queries, average QET of hot run for triple table is 14.09 min. After the partition, the average QET of hot run is reduced to 3.2 min. For our experiment, time gain is 69.83% over triple table. For most frequent queries, average QET is reduced from 12.07 min to 1.68 min resulting in time gain of 86%.

5.4 Log Scaling

Log file of various sizes are generated and is analysed using serial and parallel execution of FA and BA. This analysis is done to determine the limits of algorithms and when one should use which algorithm. Following are the observations of time taken for each algorithm to divide data as hot or cold:

Table 5.5 Time Taken to Execute FA and BA

Log Size	Time for Actual Frequency Calculation (sec)	Time for Serial Forward Algorithm (sec)	Time for Parallel Forward Algorithm (sec)	Time for Backward Algorithm (sec)
17.2M	19	20	12	32
35.7M	37	40	28	34
71.9M	68	75	60	140
145M	180	169	123	300
290M	382	318	220	622

As observed from the experiments, the parallel version of FA is found to be on average 1.4 times faster than its serial execution.

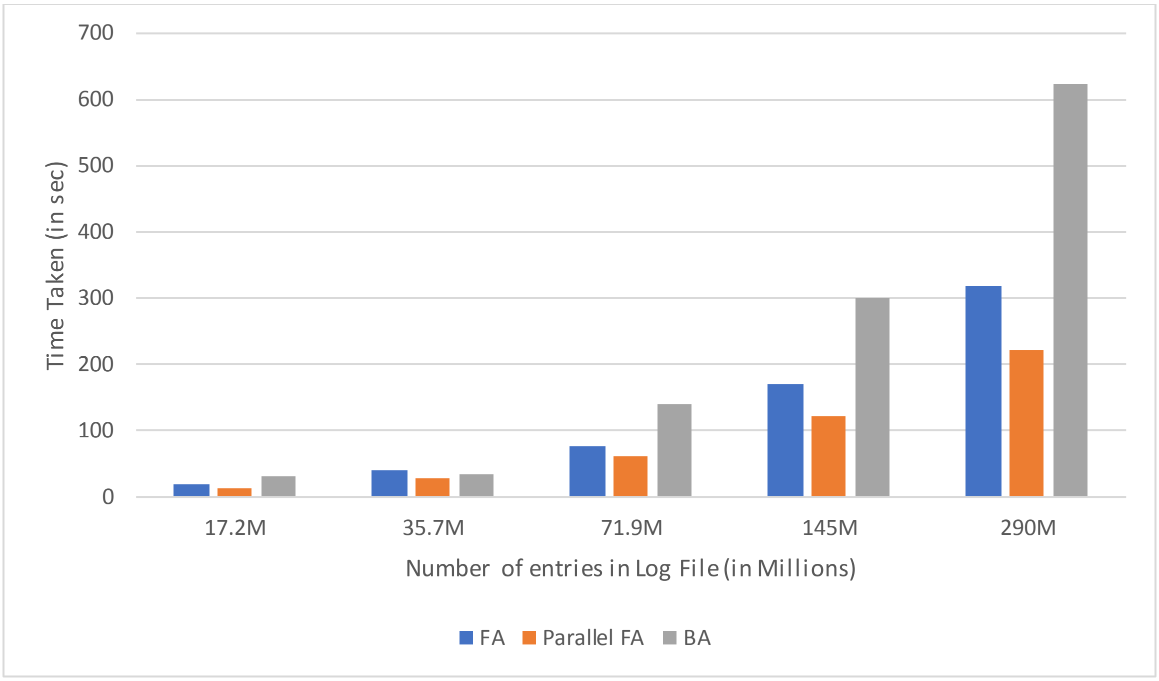


Fig. 5.2 Log Scaling

5.5 Effect of K on Hit Rate

Hit Rate is calculated to determine the difference between actual hot records and estimated hot records. As a record can be accessed more than once, a record miscalculated as cold will cause a loss in hit rate when the respective query will be executed. So, to determine hit rate we calculate the total number of times we are going to experience a loss for each miscalculated record. In case of Actual frequency calculation and Forward algorithm, sometimes more records than K% are selected as hot. It is because the excess records have same access frequency as the last record. In case of backward algorithm, access frequencies of all records are not considered, so it rejects those excess records which could be part of hot set. We have conducted the experiment to calculate hit rate in case of different values of K. The results are as shown in Table 5.6:

Table 5.6 Hit Rate in FA and BA

K (in %)	Hit Rate in Forward Algorithm (in %)	Hit Rate in Backward Algorithm (in %)	Gain in hit rate of Forward Algorithm over Backward Algorithm (in %)
0.1	99.058	0.225	98.83
2	99.8962	41.37	58.52
3	99.9918	74.43	25.56
3.2	99.9918	80.18	19.81
3.3	99.9918	82.99	16.99

3.4	99.9918	85.78	14.216
3.5	99.9918	88.55	11.44
3.7	99.9923	94.53	5.459
3.9	99.99958	99.8306	0.16247
4	99.978	99.83	0.147
4.5	99.9971	98.841	1.156
5	99.97	99.582	0.4155
5.5	100	97.41	2.59
6	100	97.84	2.156
6.5	100	98.28	1.72
7	100	98.713	1.287
7.5	100	99.147	0.8534
8	100	99.58	0.42

With increase in value of K, the hit rate of FA and BA increases. The hit rate for BA increases as the log file is read further to find out more hot records which gives more accurate results.

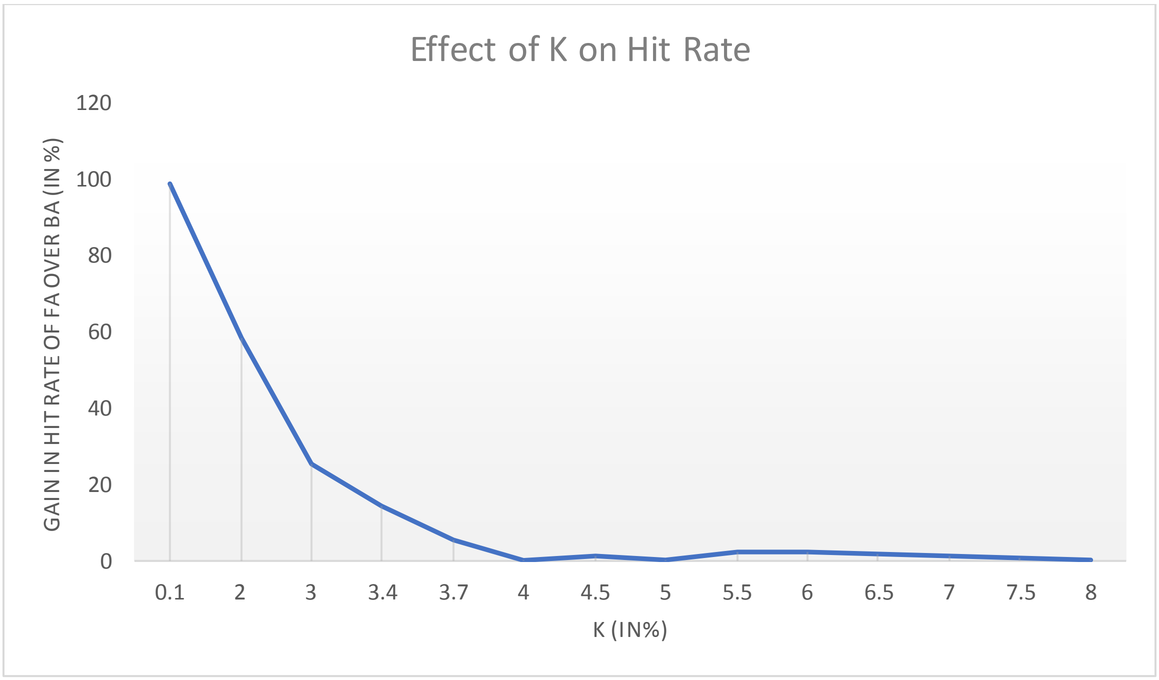


Fig. 5.3 Effect of K on Hit Rate

As shown in Fig. 5.3, the difference in hot rate of FA over BA decreases, which implies the increasing accuracy in results of BA than in lower values of K. With increase in value of K, the hot data identified by our algorithms is getting to accurate frequency estimator.

Chapter 6

Conclusion

The tremendous increase in size of IoT data has led to development of efficient methods of storage of IoT data. This thesis presents a Log Based Method LBM which partitions data into hot and cold sections. It has been found to improve query performance for IoT data in terms of query processing time which enables to build interactive IoT applications.

LBM exploits skewedness in access pattern of records in IoT dataset due to which some records are accessed frequently than the other. LBM identifies these records referred to as hot records. It uses Forward Algorithm or Backward Algorithm on a record access log file to estimate record access frequencies. It then partitions the IoT data stored as triple table into hot and cold data table which stores frequently and infrequently accessed data respectively.

Using LBM, we can identify 8% of data as hot data using which 78.6% queries are answered. The queries are found to be 67.5% faster on partitioned data than triple table. The algorithms presented in this thesis can be executed in parallel to reduce the time required for analysis which is found to be 42% faster than its serial execution.

To further accelerate the queries, the hot data partition can be moved to main memory. It can also be extended to dynamic database systems where dataset and query set can be changed dynamically. LBM can also be used for any other RDF data with known query workload.

References

[1] C. C. Aggarwal, Naveen Ashish, and Amit Sheth, "The Internet of Things: A Survey from the Data-Centric Perspective," *Managing and mining sensor data*, New York, USA, Springer, 2013, pp. 383-428.

[2] Cisco's Technology News Site. (2016, June 07). Cisco Visual Networking Index Predicts Near-Tripling of IP Traffic by 2020 [Online]. Available: https://newsroom.cisco.com/press-release-content?type=press-release&articleId=1771211.

[3] Daniel J. Abadi, Adam Marcus, Samuel Madden and Kate Hollenbach, "SW-Store: a vertically partitioned DBMS for Semantic Web data management," *The VLDB Journal—The International Journal on Very Large Data Bases 18.2*, 2009, pp. 385-406.

[4] Justin J. Levandoski and Mohamed F. Mokbel, "RDF Data-Centric Storage," *Web Services ICWS*, 2009, pp. 911-918.

[5] Padiya, Trupti, Minal Bhise, and Prashant Rajkotiya. "Data Management for Internet of Things," *Region 10 Symposium (TENSYMP) IEEE*, 2015, pp. 62-65.

[6] MahmoudiNasab, Hooran, and Sherif Sakr. "Efficient and adaptable query workload-aware management for RDF data." *International Conference on Web Information Systems Engineering. Springer*, Berlin, Heidelberg, 2010, pp. 390-399.

[7] Trupti Padiya, Jai Jai Kanwar and Minal Bhise, "Workload Aware Hybrid Partitioning," *Proceedings of the 9th Annual ACM India Conference. ACM COMPUTE*, 2016, pp. 51-58.

[8] Jai Jai Kanwar, "Hot and Cold Data Identification using Query Aware Hybrid Partitioning", M.Tech. thesis, DAIICT, Gandhinagar, Gujarat, 2015.

[9] Justin J. Levandoski, Per-Åke Larson and Radu Stoica, "Identifying hot and cold data in main-memory databases," *Data Engineering (ICDE), 2013 IEEE 29th International Conference*, pp. 26-37.

[10] Meyer, Carsten, et al. "Dynamic and Transparent Data Tiering for In-Memory Databases in Mixed Workload Environments," *ADMS, VLDB* 2015, pp. 37-48. 2015.

Publications from Thesis

- Anubha Jain, Trupti Padiya, Minal Bhise, "Log Based Method for Faster IoT Queries," accepted, to appear in IEEE TENSYMP, Cochin, Kerala, 2017.

Appendix

List of Queries

Query No.	Query Description and SQL Query
1	Find temperature observation for sensor '<http://knoesis.wright.edu/ssw/System_MVEM7>'. select L5.obj from lod8 L1, lod8 L2, lod8 L3, lod8 L4, lod8 L5 where L1.sub = '<http://knoesis.wright.edu/ssw/System_MVEM7>' and L1.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#generatedObservation>' and L2.sub = L1.obj and L2.pred = '<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>' and L2.obj = '<http://knoesis.wright.edu/ssw/ont/weather.owl#TemperatureObservation>' and L3.sub = L2.sub and L3.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#result>' and L4.obj = '<http://knoesis.wright.edu/ssw/ont/weather.owl#fahrenheit>' and L4.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#uom>' and L4.sub = L3.obj and L5.sub = L4.sub and L5.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#floatValue>';
2	Find temperature observation for sensor '<http://knoesis.wright.edu/ssw/System_NFFI1>' select L5.obj from lod8 L1, lod8 L2, lod8 L3, lod8 L4, lod8 L5 where L1.sub = '<http://knoesis.wright.edu/ssw/System_NFFI1>' and L1.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#generatedObservation>' and L2.sub = L1.obj and L2.pred = '<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>' and L2.obj = '<http://knoesis.wright.edu/ssw/ont/weather.owl#TemperatureObservation>' and L3.sub = L2.sub and L3.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#result>' and L4.obj = '<http://knoesis.wright.edu/ssw/ont/weather.owl#fahrenheit>' and L4.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#uom>' and L4.sub = L3.obj and L5.sub = L4.sub and L5.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#floatValue>';
3	Find all rainfall observations. select L4.obj from lod8 L1, lod8 L2, lod8 L3, lod8 L4 where L1.obj = '<http://knoesis.wright.edu/ssw/ont/weather.owl#RainfallObservation>' and L1.pred = '<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>' and L2.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#result>' and L2.sub = L1.sub and L3.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#uom>'

	and L3.obj = '<http://knoesis.wright.edu/ssw/ont/weather.owl#centimeters>' and L3.sub = L2.obj and L4.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#floatValue>' and L4.sub = L3.sub;
4	Find humidity at location '<http://knoesis.wright.edu/ssw/point_ KTIP'. select L6.obj from lod8 L1, lod8 L2, lod8 L3, lod8 L4, lod8 L5, lod8 L6 where L1.obj = '<http://knoesis.wright.edu/ssw/point_ KTIP>' and L1.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#processLocation>' and L2.sub = L1.sub and L2.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#generatedObservation>' and L3.sub = L2.obj and L3.pred = '<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>' and L3.obj = '<http://knoesis.wright.edu/ssw/ont/weather.owl#RelativeHumidityObservation>' and L4.sub = L3.sub and L4.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#result>' and L5.obj = '<http://knoesis.wright.edu/ssw/ont/weather.owl#percent>' and L5.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#uom>' and L5.sub = L4.obj and L6.sub = L5.sub and L6.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#floatValue>';
5	Find humidity at location '<http://knoesis.wright.edu/ssw/point_KTMH>'. select L6.obj from lod8 L1, lod8 L2, lod8 L3, lod8 L4, lod8 L5, lod8 L6 where L1.obj = 'http://knoesis.wright.edu/ssw/point_ KTMH' and L1.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#processLocation>' and L2.sub = L1.sub and L2.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#generatedObservation>' and L3.sub = L2.obj and L3.pred = '<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>' and L3.obj = '<http://knoesis.wright.edu/ssw/ont/weather.owl#RelativeHumidityObservation>' and L4.sub = L3.sub and L4.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#result>' and L5.obj = '<http://knoesis.wright.edu/ssw/ont/weather.owl#percent>' and L5.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#uom>' and L5.sub = L4.obj and L6.sub = L5.sub and L6.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#floatValue>';
6	Find rainfall observation where relative humidity is less than '"32.0"^^<http://www.w3.org/2001/XMLSchema#float>'. select T4.obj from lod8 T1, lod8 T2, lod8 T3, lod8 T4, lod8 T5 where T1.obj = '<http://knoesis.wright.edu/ssw/ont/weather.owl#RainfallObservation>' and T1.pred = '<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>' and T2.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#result>'

	and T2.sub = T1.sub and T3.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#uom>' and T3.obj = '<http://knoesis.wright.edu/ssw/ont/weather.owl#centimeters>' and T3.sub = T2.obj and T4.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#floatValue>' and T4.sub = T3.sub and T5.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#generatedObservation>' and T1.sub = T5.obj and T5.sub IN (*select L2.sub from lod8 L2, lod8 L3, lod8 L4, lod8 L5, lod8 L6 where L2.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#generatedObservation>' and L3.sub = L2.obj* and L3.pred = '<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>' and L3.obj = '<http://knoesis.wright.edu/ssw/ont/weather.owl#RelativeHumidityObservation>' and L4.sub = L3.sub and L4.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#result>' and L5.obj = '<http://knoesis.wright.edu/ssw/ont/weather.owl#percent>' and L5.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#uom>' and L5.sub = L4.obj and L6.sub = L5.sub and L6.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#floatValue>' and L6.obj < '"32.0"^^<http://www.w3.org/2001/XMLSchema#float>'");
7	Find all sampling time for wind observation on date : 2004-08-10. select L2.obj from lod8 L1, lod8 L2, lod8 L3 where L1.obj = '<http://knoesis.wright.edu/ssw/ont/weather.owl#WindSpeedObservation>' and L1.pred = '<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>' and L2.sub = L1.sub and L2.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#samplingTime>' and L3.sub = L2.obj and L3.pred = '<http://www.w3.org/2006/time#inXSDDateTime>' and L3.obj like '"2004-08-10%';
8	Find wind observation at location "<http://knoesis.wright.edu/ssw/point_ KUGN >". select L6.obj from lod8 L1, lod8 L2, lod8 L3, lod8 L4, lod8 L5, lod8 L6 where L1.obj = '<http://knoesis.wright.edu/ssw/point_ KUGN>' and L1.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#processLocation>' and L2.sub = L1.sub and L2.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#generatedObservation>' and L3.sub = L2.obj and L3.pred = '<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>' and L3.obj = '<http://knoesis.wright.edu/ssw/ont/weather.owl#WindSpeedObservation>' and L4.sub = L3.sub

	and L4.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#result>' and L5.obj = '<http://knoesis.wright.edu/ssw/ont/weather.owl#milesPerHour>' and L5.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#uom>' and L5.sub = L4.obj and L6.sub = L5.sub and L6.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#floatValue>';
9	Find wind observation at location "<http://knoesis.wright.edu/ssw/point_ MKGC1>". select L6.obj from lod8 L1, lod8 L2, lod8 L3, lod8 L4, lod8 L5, lod8 L6 where L1.obj = '<http://knoesis.wright.edu/ssw/point_ MKGC1>' and L1.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#processLocation>' and L2.sub = L1.sub and L2.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#generatedObservation>' and L3.sub = L2.obj and L3.pred = '<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>' and L3.obj = '<http://knoesis.wright.edu/ssw/ont/weather.owl#WindSpeedObservation>' and L4.sub = L3.sub and L4.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#result>' and L5.obj = '<http://knoesis.wright.edu/ssw/ont/weather.owl#milesPerHour>' and L5.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#uom>' and L5.sub = L4.obj and L6.sub = L5.sub and L6.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#floatValue>';
10	Find average rainfall at place "xyz" on 2004-08-12 and 2004-08-13. (select L4.OID, clock_timestamp() from lod8 L1, lod8 L2, lod8 L3, lod8 L4, lod8 L5, lod8 L6 where L1.obj = '<http://knoesis.wright.edu/ssw/ont/weather.owl#RainfallObservation>' and L1.pred = '<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>' and L2.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#result>' and L2.sub = L1.sub and L3.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#uom>' and L3.obj = '<http://knoesis.wright.edu/ssw/ont/weather.owl#centimeters>' and L3.sub = L2.obj and L4.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#floatValue>' and L4.sub = L3.sub and L6.obj like '"2004-08-12%' and L6.pred = '<http://www.w3.org/2006/time#inXSDDateTime>' and L5.obj = L6.sub and L5.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#samplingTime>' and L5.sub = L1.sub) union (select T4.OID, clock_timestamp() from lod8 T1, lod8 T2, lod8 T3, lod8 T4, lod8 T5, lod8 T6 where T1.obj = '<http://knoesis.wright.edu/ssw/ont/weather.owl#RainfallObservation>' and T1.pred = '<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>'

	and T2.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#result>' and T2.sub = T1.sub and T3.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#uom>' and T3.obj = '<http://knoesis.wright.edu/ssw/ont/weather.owl#centimeters>' and T3.sub = T2.obj and T4.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#floatValue>' and T4.sub = T3.sub and T6.obj like '"2004-08-13%' and T6.pred = '<http://www.w3.org/2006/time#inXSDDateTime>' and T5.obj = T6.sub and T5.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#samplingTime>' and T5.sub = T1.sub);
11	Find max wind gust at place "<http://knoesis.wright.edu/ssw/point_ LICC1>". select L6.obj from lod8 L1, lod8 L2, lod8 L3, lod8 L4, lod8 L5, lod8 L6 where L1.obj = '<http://knoesis.wright.edu/ssw/point_ LICC1>' and L1.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#processLocation>' and L2.sub = L1.sub and L2.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#generatedObservation>' and L3.sub = L2.obj and L3.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#observedProperty>' and L3.obj = '<http://knoesis.wright.edu/ssw/ont/weather.owl#_WindGust>' and L4.sub = L3.sub and L4.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#result>' and L5.obj = '<http://knoesis.wright.edu/ssw/ont/weather.owl#milesPerHour>' and L5.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#uom>' and L5.sub = L4.obj and L6.sub = L5.sub and L6.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#floatValue>';
10	Find max wind gust at place "<http://knoesis.wright.edu/ssw/point_ LOF>". select L6.obj from lod8 L1, lod8 L2, lod8 L3, lod8 L4, lod8 L5, lod8 L6 where L1.obj = '<http://knoesis.wright.edu/ssw/point_ LOF>' and L1.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#processLocation>' and L2.sub = L1.sub and L2.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#generatedObservation>' and L3.sub = L2.obj and L3.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#observedProperty>' and L3.obj = '<http://knoesis.wright.edu/ssw/ont/weather.owl#_WindGust>' and L4.sub = L3.sub and L4.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#result>' and L5.obj = '<http://knoesis.wright.edu/ssw/ont/weather.owl#milesPerHour>' and L5.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#uom>' and L5.sub = L4.obj and L6.sub = L5.sub

	and L6.pred = '<http://knoesis.wright.edu/ssw/ont/sensor- observation.owl#floatValue>';
11	Find dew point at location "<http://knoesis.wright.edu/ssw/point_ MBOU>". select L6.obj from lod8 L1, lod8 L2, lod8 L3, lod8 L4, lod8 L5, lod8 L6 where L1.obj = '<http://knoesis.wright.edu/ssw/point_ MBOU>' and L1.pred = '<http://knoesis.wright.edu/ssw/ont/sensor- observation.owl#processLocation>' and L2.sub = L1.sub and L2.pred = '<http://knoesis.wright.edu/ssw/ont/sensor- observation.owl#generatedObservation>' and L3.sub = L2.obj and L3.pred = '<http://knoesis.wright.edu/ssw/ont/sensor- observation.owl#observedProperty>' and L3.obj = '<http://knoesis.wright.edu/ssw/ont/weather.owl#_DewPoint>' and L4.sub = L3.sub and L4.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#result>' and L5.obj = '<http://knoesis.wright.edu/ssw/ont/weather.owl#fahrenheit>' and L5.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#uom>' and L5.sub = L4.obj and L6.sub = L5.sub and L6.pred = '<http://knoesis.wright.edu/ssw/ont/sensor- observation.owl#floatValue>';
12	Find dew point at location "<http://knoesis.wright.edu/ssw/point_ MFLT>". select L6.obj from lod8 L1, lod8 L2, lod8 L3, lod8 L4, lod8 L5, lod8 L6 where L1.obj = '<http://knoesis.wright.edu/ssw/point_ MFLT>' and L1.pred = '<http://knoesis.wright.edu/ssw/ont/sensor- observation.owl#processLocation>' and L2.sub = L1.sub and L2.pred = '<http://knoesis.wright.edu/ssw/ont/sensor- observation.owl#generatedObservation>' and L3.sub = L2.obj and L3.pred = '<http://knoesis.wright.edu/ssw/ont/sensor- observation.owl#observedProperty>' and L3.obj = '<http://knoesis.wright.edu/ssw/ont/weather.owl#_DewPoint>' and L4.sub = L3.sub and L4.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#result>' and L5.obj = '<http://knoesis.wright.edu/ssw/ont/weather.owl#fahrenheit>' and L5.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#uom>' and L5.sub = L4.obj and L6.sub = L5.sub and L6.pred = '<http://knoesis.wright.edu/ssw/ont/sensor- observation.owl#floatValue>';
13	Find Id and location of a sensor "<http://knoesis.wright.edu/ssw/System_ A20>". select L1.obj from lod8 L1, lod8 L2 where L1.sub = '<http://knoesis.wright.edu/ssw/System_ A20>' and L1.pred = '<http://knoesis.wright.edu/ssw/ont/sensor- observation.owl#processLocation>' and L2.sub = L1.sub and L2.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#ID>';
14	Find lat/alt information for a sensor "<http://knoesis.wright.edu/ssw/System_CYNE>".

	select L2.obj, L3.obj from lod8 L1, lod8 L2, lod8 L3 where L1.sub = '<http://knoesis.wright.edu/ssw/System_CYNE>' and L1.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#processLocation>' and L2.sub = L1.obj and L2.pred = '<http://www.w3.org/2003/01/geo/wgs84_pos#alt>' and L3.sub = L1.obj and L3.pred = '<http://www.w3.org/2003/01/geo/wgs84_pos#lat>';
15	Find sensors at located near "<http://knoesis.wright.edu/ssw/LocatedNearRelLPOC1>". select distinct L2.obj from lod8 L1, lod8 L2 where L1.obj = '<http://knoesis.wright.edu/ssw/LocatedNearRelLPOC1>' and L1.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#hasLocatedNearRel>' and L2.sub = L1.sub and L2.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#ID>';
16	List sensors that generate wind observation. select L1.sub from lod8 L1 where L1.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#parameter>' and L1.obj = '<http://knoesis.wright.edu/ssw/ont/weather.owl#_WindSpeed>';
17	List sensors that generate dew point measurement. select L1.sub from lod8 L1 where L1.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#parameter>' and L1.obj = '<http://knoesis.wright.edu/ssw/ont/weather.owl#_DewPoint>';
18	Find geoname, point of a "System_KILN". select L2.obj from lod8 L1, lod8 L2 where L1.sub = '<http://knoesis.wright.edu/ssw/System_KILN>' and L1.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#hasLocatedNearRel>' and L2.sub = L1.sub and L2.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#processLocation>';
19	Find sensors located near "<http://knoesis.wright.edu/ssw/LocatedNearRelMAMF >" with freezing temperature. select distinct L1.sub from lod8 L1, lod8 L2, lod8 L3, lod8 L4, lod8 L5, lod8 L6 where L1.obj = '<http://knoesis.wright.edu/ssw/LocatedNearRelMAMF >' and L1.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#hasLocatedNearRel>' and L2.sub = L1.sub and L2.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#generatedObservation>' and L3.sub = L2.obj and L3.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#observedProperty>' and L3.obj = '<http://knoesis.wright.edu/ssw/ont/weather.owl#_AirTemperature>' and L4.sub = L3.sub and L4.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#result>' and L5.obj = '<http://knoesis.wright.edu/ssw/ont/weather.owl#fahrenheit>'

	and L5.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#uom>' and L5.sub = L4.obj and L6.sub = L5.sub and L6.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#floatValue>' and L6.obj < '"32.0"^^<http://www.w3.org/2001/XMLSchema#float>';
20	Find rainfall observation where relative humidity is greater than '"50.0"^^<http://www.w3.org/2001/XMLSchema#float>'. select T4.obj from lod8 T1, lod8 T2, lod8 T3, lod8 T4, lod8 T5 where T1.obj = '<http://knoesis.wright.edu/ssw/ont/weather.owl#RainfallObservation>' and T1.pred = '<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>' and T2.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#result>' and T2.sub = T1.sub and T3.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#uom>' and T3.obj = '<http://knoesis.wright.edu/ssw/ont/weather.owl#centimeters>' and T3.sub = T2.obj and T4.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#floatValue>' and T4.sub = T3.sub and T5.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#generatedObservation>' and T1.sub = T5.obj and T5.sub IN (select L2.sub from lod8 L2, lod8 L3, lod8 L4, lod8 L5, lod8 L6 where L2.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#generatedObservation>' and L3.sub = L2.obj and L3.pred = '<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>' and L3.obj = '<http://knoesis.wright.edu/ssw/ont/weather.owl#RelativeHumidityObservation>' and L4.sub = L3.sub and L4.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#result>' and L5.obj = '<http://knoesis.wright.edu/ssw/ont/weather.owl#percent>' and L5.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#uom>' and L5.sub = L4.obj and L6.sub = L5.sub and L6.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#floatValue>' and L6.obj > '"50.0"^^<http://www.w3.org/2001/XMLSchema#float>'");
21	find average wind speed at place "point ODT15". select distinct(l6.obj) from lodtriples l1, lodtriples l2, lodtriples l3, lodtriples l4, lodtriples l5, lodtriples l6 where l1.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#ID>' and l1.obj like '% ODT15%' and l2.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#generatedObservation>' and l1.sub = l2.sub and l3.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#parameter>' and l1.sub like l3.sub and l3.obj = '<http://knoesis.wright.edu/ssw/ont/weather.owl#_WindSpeed>' and l4.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#result>' and l4.sub = l2.obj

	and l5.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#uom>' and l5.obj = '<http://knoesis.wright.edu/ssw/ont/weather.owl#milesPerHour>' and l5.sub = l4.obj and l6.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#floatValue>' and l4.obj = l6.sub;
22	Find observations when wind speed value is 9. select L1.sub,L1.obj from lod8 L1, lod8 L2, lod8 L3 where L1.pred like '<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>' and L1.sub like '<http://knoesis.wright.edu/ssw/Observation_WindSpeed%' and L2.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#floatValue>' and L2.obj like '"9.0"^^<http://www.w3.org/2001/XMLSchema#float>' and L3.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#result>' and L1.sub=L3.sub and L2.sub=L3.obj;
23	Find sensors where relative humidity observed is "25.0"^^http://www.w3.org/2001/XMLSchema#float. select L1.sub,L2.obj,L2.sub,L2.obj from "LODTriples" L1, "LODTriples" L2, "LODTriples" L3 where L1.pred like '<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>' and L1.obj like '<http://knoesis.wright.edu/ssw/ont/weather.owl#RelativeHumidityObservation>' and L2.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#floatValue>' and L2.obj like '"25.0"^^<http://www.w3.org/2001/XMLSchema#float>' and L3.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#result>' and L2.sub = L3.obj and L1.sub=L3.sub;
24	Find sysstem name and location with their ids. select L1.sub,L1.obj,L2.obj from "LODTriples" L1, "LODTriples" L2 where L1.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#ID>' and L2.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#processLocation>' and L1.sub = L2.sub;
25	Find all the wind speed observations. select L1.sub from lod8 L1 where L1.obj like '<http://knoesis.wright.edu/ssw/ont/weather.owl#WindSpeedObservation>' and L1.pred = '<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>';
26	Find sampling time of all snowfall observations. select L2.obj from lod8 L1, lod8 L2 where L1.obj = '<http://knoesis.wright.edu/ssw/ont/weather.owl#SnowfallObservation>' and L1.pred = '<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>' and L2.sub = L1.sub and L2.pred = '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#samplingTime>';

27	Find all visibility observations. select L1.sub from lod8 L1 where L1.obj like '<http://knoesis.wright.edu/ssw/ont/weather.owl#VisibilityObservation>' and L1.pred = '<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>';
28	Find all values of rainfall observations. select L1.OID as oid1,L2.OID as oid2,L3.OID as oid3 from lod8 L1, lod8 L2, lod8 L3 where L1.pred like '<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>' and L1.sub like '<http://knoesis.wright.edu/ssw/Observation_Precipitation%' and L2.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#floatValue>' and L3.pred like '<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#result>' and L1.sub=L3.sub and L2.sub=L3.obj;