NATIONAL OPEN UNIVERSITY OF NIGERIA

SCHOOL OF SCIENCE AND TECHNOLOGY

COURSE CODE: DAM 212

COURSE TITLE: DATABASE LABORATORY

DAM 212: **DATABASE LABORATORY**

COURSE CODE: DAM 212

COURSE TITLE: **DATABASE LABORATORY**

Course Developer/Writer: DR. A. F. Adekoya B.Sc, M.Sc, MBA, PGDE, Ph.D.

Department of Computer Science

College of Natural Sciences

Federal University of Agriculture

Abeokuta, Nigeria.

Course Editor:

Programme Leader: National Open University of Nigeria

Course Coordinator: National Open University of Nigeria

NATIONAL OPEN UNIVERSITY OF NIGERIA

DAM 212: DATABASE LABORATORY

CONTENTS PAGE

DAM 212: DATABASE LABORATORY

Module 1: Performance Tuning on Relational Database

Unit 1: Basic Concept and Definition

1.0 Introduction

1.1 Objectives

1.2 Definitions

1.3 History of Database

1.4 Database Models

1.5 Database terminologies

1.6 Application of Database

Module 1: Performance Tuning on Relational Database

Unit 1: Basic Concept and Definition

1.0 Introduction

This is a self-paced laboratory course exploring topics in data management. Students will devise efficient solutions to real-world data management problems on realistic data sets. Performance tuning of relational databases, querying graph-structured databases, querying streaming databases, indexing high-dimensional data, managing uncertainty in databases, information retrieval in databases, and implementation of database internals. Students will use an open-source database management system and a programming language API (like JDBC/ODBC, MS-Access).

1.1 Objectives

1.2 Concept Definition

Database: A single collection of information to which the user has access through a collection of programs. It refers to any file that can be accessed by a key other than its ordering key.

Database Management System : A group or collection of programs that give the user access to a collection of information.

Delimiter: Any character or symbol used to separate data in a record.

Domain: A collection of fields of the same type.

Entity: Something about which data is recorded.

Entity Set: A collection of information.

1.3 History of Database

The earliest known use of the term '***data base'*** was in June 1963, when the System Development Corporation sponsored a symposium under the title *Development and Management of a Computer-centered Data Base*. **Database** as a single word became common in Europe in the early 1970s and by the end of the decade it was being used in major American newspapers.

(**Databank**, a comparable term, had been used in the *Washington Post* newspaper as early as 1966.)

The first database management systems were developed in the 1960s. A pioneer in the field was Charles Bachman. Bachman's early papers show that his aim was to make more effective use of the new direct access storage devices becoming available. Until then, data processing had been based on punched cards and magnetic tape, so that serial processing was the dominant activity. Two key data models arose at this time: CODASYL developed the network model based on Bachman's ideas, and (apparently independently) the hierarchical model was used in a system developed by North American Rockwell, later adopted by IBM as the cornerstone of their IMS product.

The relational model was proposed by Professor E. F. Codd in 1970. He criticized existing models for confusing the abstract description of information structure with descriptions of physical access mechanisms. For a long while, however, the relational model remained of academic interest only. While CODASYL systems and IMS were conceived as practical engineering solutions taking account of the technology as it existed at the time, the relational model took a much more theoretical perspective, arguing (correctly) that hardware and software technology would catch up in time. Among the first implementations were Michael Stonebraker's Ingres at Berkeley, and the SystemR project at IBM. Both of these were research prototypes, announced during 1976. The first commercial products, Oracle and DB2, did not appear until around 1980. The first successful database product for microcomputers was dBASE for the CP/M and PC-DOS/MS-DOS operating systems.

During the 1980s, research activity focused on distributed database systems and database machines, but these developments had little effect on the market. Another important theoretical idea was the Functional Data Model, but apart from some specialized applications in genetics, molecular biology, and fraud investigation, the world took little notice. In the 1990s, attention shifted to object-oriented databases. These had some success in fields where it was necessary to handle more complex data than relational systems could easily cope with, such as spatial databases, engineering data (including software engineering repositories), and multimedia data. Some of these ideas were adopted by the relational vendors, who integrated new features into their products as a result.

In the 2000s, the fashionable area for innovation is the XML database. As with object databases, this has spawned a new collection of startup companies, but at the same time the key ideas are being integrated into the established relational products. XML databases aim to remove the traditional divide between documents and data, allowing all of an organization's information resources to be held in one place, whether they are highly structured or not.

1.4 Database Models

Various techniques are used to model data structure. Most database systems are built around one particular data model, although it is increasingly common for products to offer support for more than one model. For any one logical model various physical implementations may be possible, and most products will offer the user some level of control in tuning the physical implementation, since the choices that are made have a significant effect on performance. An

example of this is the relational model: all serious implementations of the relational model allow the creation of indexes which provide fast access to rows (records) in a table if the values of certain columns are known.

A data model is not just a way of structuring data; it also defines a set of operations that can be performed on the data. The relational model, for example, defines operations such as select, project, and joins. Although these operations may not be explicit in a particular query language, they provide the foundation on which a query language is built.

A. Flat model

This may not strictly qualify as a data model, as defined above. The flat (or table) model consists of a single, two-dimensional array of data elements, where all members of a given column are assumed to be similar values, and all members of a row are assumed to be related to one another. For instance, columns for name and password that might be used as a part of a system security database. Each row would have the specific password associated with an individual user. Columns of the table often have a type associated with them, defining them as character data, date or time information, integers, or floating point numbers. This model is, incidentally, a basis of the spreadsheet.

B. Hierarchical model

In a hierarchical model, data is organized into a tree-like structure, implying a single upward link in each record to describe the nesting, and a sort field to keep the records in a particular order in each same-level list. Hierarchical structures were widely used in the early mainframe database management systems, such as the Information Management System (IMS) by IBM, and now describe the structure of XML documents. This structure allows one 1:N relationship between two types of data. This structure is very efficient to describe many relationships in the real world; recipes, table of contents, ordering of paragraphs/verses, any nested and sorted information. However, the hierarchical structure is inefficient for certain database operations when a full path (as opposed to upward link and sort field) is not also included for each record.

C. Network model

The network model (defined by the CODASYL specification) organizes data using two fundamental constructs, called *records* and *sets*. Records contain fields (which may be organized hierarchically, as in the programming language COBOL). Sets (not to be confused with mathematical sets) define one-to-many relationships between records: one owner, many members. A record may be an owner in any number of sets, and a member in any number of sets. The operations of the network model are navigational in style: a program maintains a current position, and navigates from one record to another by following the relationships in which the record participates. Records can also be located by supplying key values. Although it is not an essential feature of the model, network databases generally implement the set relationships by means of pointers that directly address the location of a record on disk. This gives excellent retrieval performance, at the expense of operations such as database loading and reorganization.

D. Relational model

The relational model was introduced in an academic paper by Prof. E. F. Codd in 1970 as a way to make database management systems more independent of any particular application. It is a mathematical model defined in terms of predicate logic and set theory.
The products that are generally referred to as relational databases in fact implement a model that is only an approximation to the mathematical model defined by Codd. The data structures in these products are tables, rather than relations: the main differences being that tables can contain duplicate rows, and that the rows (and columns) can be treated as being ordered. The same criticism applies to the SQL language which is the primary interface to these products. There has been considerable controversy, mainly due to Codd himself, as to whether it is correct to describe SQL implementations as "relational": but the fact is that the world does so, and the following description uses the term in its popular sense.

A relational database contains multiple tables, each similar to the one in the "flat" database model. Relationships between tables are not defined explicitly; instead, *keys* are used to match up rows of data in different tables. A key is a collection of one or more columns (fields) in one table whose values match corresponding columns in other tables: for example, an *Employee* table may contain a column named *Location* which contains a value that matches the key of a *Location* table. Any column can be a key, or multiple columns can be grouped together into a single key. It is not necessary to define all the keys in advance; a column can be used as a key even if it was not originally intended to be one. A key that can be used to uniquely identify a row in a table is called a *unique key*. Typically one of the unique keys is the preferred way to refer to a row; this is defined as the table's primary key.

A key that has an external, real-world meaning (such as a person's name, a book's ISBN, or a car's serial number) is sometimes called a "natural" key. If no natural key is suitable (think of the many people named *Brown*), an arbitrary key can be assigned (such as by giving employees ID numbers). In practice, most databases have both generated and natural keys, because generated keys can be used internally to create links between rows that cannot break, while natural keys can be used, less reliably, for searches and for integration with other databases. (For example, records in two independently developed databases could be matched up by social security number, except when the social security numbers are incorrect, missing, or have changed.)

Relational operations
Users (or programs) request data from a relational database by sending it a query that is written in a special language, usually a dialect of SQL. Although SQL was originally intended for end-users, it is much more common for SQL queries to be embedded into software that provides an easier user interface. Many web sites, such as Wikipedia, perform SQL queries when generating pages. In response to a query, the database returns a result set, which is just a list of rows containing the answers. The simplest query is just to return all the rows from a table, but more often, the rows are filtered in some way to return just the answer wanted.

Often, data from multiple tables are combined into one, by doing a join. Conceptually, this is done by taking all possible combinations of rows (the Cartesian product), and then filtering out everything except the answer. In practice, relational database management systems rewrite ("optimize") queries to perform faster, using a variety of techniques.

There are a number of relational operations in addition to join. These include:

- project (the process of eliminating some of the columns);
- restrict (the process of eliminating some of the rows);
- union (a way of combining two tables with similar structures);
- difference (which lists the rows in one table that are not found in the other);
- intersect (which lists the rows found in both tables); and
- product (mentioned above, which combines each row of one table with each row of the other).

Depending on which other sources you consult, there are a number of other operators - many of which can be defined in terms of those listed above. These include *semi-join*, outer operators such as *outer join* and *outer union*, and various forms of division. Then there are operators to rename columns, and summarizing or aggregating operators, and if you permit relation values as attributes (RVA - relation-valued attribute), then operators such as group and ungroup. The SELECT statement in SQL serves to handle all of these except for the group and ungroup operators.

The flexibility of relational databases allows programmers to write queries that were not anticipated by the database designers. As a result, relational databases can be used by multiple applications in ways the original designers did not foresee, which is especially important for databases that might be used for decades. This has made the idea and implementation of relational databases very popular with businesses.

E. Dimensional model

The dimensional model is a specialized adaptation of the relational model used to represent data in data warehouses in a way that data can be easily summarized using *On–Line Analytical Processing* (OLAP) queries. In the dimensional model, a database consists of a single large table of facts that are described using dimensions and measures. A dimension provides the context of a fact (such as who participated, when and where it happened, and its type) and is used in queries to group related facts together. Dimensions tend to be discrete and are often hierarchical; for example, the location might include the building, state, and country. A measure is a quantity describing the fact, such as revenue. It's important that measures can be meaningfully aggregated - for example, the revenue from different locations can be added together. In an OLAP query, dimensions are chosen and the facts are grouped and added together to create a summary.

The dimensional model is often implemented on top of the relational model using a star schema, consisting of one table containing the facts and surrounding tables containing the dimensions. Particularly complicated dimensions might be represented using multiple tables, resulting in a snowflake schema. A data warehouse can contain multiple star schemas that share dimension tables, allowing them to be used together. Coming up with a standard set of dimensions is an important part of dimensional modeling.

F. Object database models

In recent years, the object-oriented paradigm has been applied to database technology, creating a new programming model known as object databases. These databases attempt to bring the database world and the application programming world closer together, in particular by ensuring that the database uses the same type of system as the application program. This aims to avoid the

overhead (sometimes referred to as the *impedance mismatch*) of converting information between its representation in the database (for example as rows in tables) and its representation in the application program (typically as objects). At the same time object databases attempt to introduce the key ideas of object programming, such as encapsulation and polymorphism, into the world of databases.

A variety of these ways have been tried for storing objects in a database. Some products have approached the problem from the application programming end, by making the objects manipulated by the program persistent. This also typically requires the addition of some kind of query language, since conventional programming languages do not have the ability to find objects based on their information content. Others have attacked the problem from the database end, by defining an object-oriented data model for the database, and defining a database programming language that allows full programming capabilities as well as traditional query facilities.

Object databases suffered because of a lack of standardization; nevertheless, object databases have been used successfully in many applications: usually specialized applications such as engineering databases or molecular biology databases rather than mainstream commercial data processing. However, object database ideas were picked up by the relational vendors and influenced extensions made to these products and indeed to the SQL language.

1.5 Database terminologies

a. Indexing

All kinds of databases can take advantage of indexing to increase their speed, and this technology has advanced tremendously since its early uses in the 1960s and 1970s. The most common kind of index is a sorted list of the contents of some particular table column, with pointers to the row associated with the value. An index allows a set of table rows matching some criterion to be located quickly. Various methods of indexing are commonly used; B-trees, hashes, and linked lists are all common indexing techniques.

Relational DBMSs have the advantage that indexes can be created or dropped without changing existing applications making use of it. The database chooses between many different strategies based on which one it estimates will run the fastest. Relational DBMSs utilize many different algorithms to compute the result of an SQL statement. The RDBMS will produce a plan of how to execute the query, which is generated by analyzing the run times of the different algorithms and selecting the quickest. Some of the key algorithms that deal with joins are Nested Loops Join, Sort-Merge Join and Hash Join.

b. Transactions and concurrency

In addition to their data model, most practical databases ("transactional databases") attempt to enforce a database transaction model that has desirable data integrity properties. Ideally, the database software should enforce the ***ACID*** rules, summarized here:

- **A**tomicity: Either all the tasks in a transaction must be done, or none of them. The transaction must be completed, or else it must be undone (rolled back).

- **C**onsistency: Every transaction must preserve the integrity constraints — the declared consistency rules — of the database. It cannot place the data in a contradictory state.
- **I**solation: Two simultaneous transactions cannot interfere with one another. Intermediate results within a transaction are not visible to other transactions.
- **D**urability: Completed transactions cannot be aborted later or their results discarded. They must persist through (for instance) restarts of the DBMS after crashes

In practice, many DBMS's allow most of these rules to be selectively relaxed for better performance. Concurrency control is a method used to ensure that transactions are executed in a safe manner and follow the ACID rules. The DBMS must be able to ensure that only serializable, recoverable schedules are allowed, and that no actions of committed transactions are lost while undoing aborted transactions.

c. Replication

Replication of databases is closely related to transactions. If a database can log its individual actions, it is possible to create a duplicate of the data in real time. The duplicate can be used to improve performance or availability of the whole database system. Common replication concepts include:

- Master/Slave Replication: All write requests are performed on the master and then replicated to the slaves
- Quorum: The result of Read and Write requests is calculated by querying a "majority" of replicas.
- Multimaster: Two or more replicas sync each other via a transaction identifier.

1.6 APPLICATIONS OF DATABASES

Databases are used in many applications, spanning virtually the entire range of computer software. Databases are the preferred method of storage for large multi-user applications, where coordination between many users is needed. Even individual users find them convenient, though, and many electronic mail programs and personal organizers are based on standard database technology. Software database drivers are available for most database platforms so that application software can use a common application programming interface (API) to retrieve the information stored in a database. Two commonly used database Applications Programming Interfaces (APIs) are Java Database Connectivity (JDBC)and Open Database Connectivity (ODBC).

Module 1: Performance Tuning on Relational Database

Unit 2: Relational database system design

2.0 Relational Database Overview

A database is a means of storing information in such a way that information can be retrieved from it. In simplest terms, a relational database is one that presents information in tables with rows and columns. A table is referred to as a relation in the sense that it is a collection of objects of the same type (rows). Data in a table can be related according to common keys or concepts, and the ability to retrieve related data from a table is the basis for the term relational database. A Database Management System (DBMS) handles the way data is stored, maintained, and retrieved. In the case of a relational database, a Relational Database Management System (RDBMS) performs these tasks. DBMS as used in this book is a general term that includes RDBMS.

2.1 Integrity Rules

Relational tables follow certain integrity rules to ensure that the data they contain stay accurate and are always accessible. First, the rows in a relational table should all be distinct. If there are duplicate rows, there can be problems resolving which of two possible selections is the correct one. For most DBMSs, the user can specify that duplicate rows are not allowed, and if that is done, the DBMS will prevent the addition of any rows that duplicate an existing row.

A second integrity rule of the traditional relational model is that column values must not be repeating groups or arrays. A third aspect of data integrity involves the concept of a null value. A database takes care of situations where data may not be available by using a null value to indicate that a value is missing. It does not equate to a blank or zero. A blank is considered equal to another blank, a zero is equal to another zero, but two null values are not considered equal.

When each row in a table is different, it is possible to use one or more columns to identify a particular row. This unique column or group of columns is called a primary key. Any column that is part of a primary key cannot be null; if it were, the primary key containing it would no longer be a complete identifier. This rule is referred to as entity integrity.

Table 2.1 illustrates some of these relational database concepts. It has five columns and six rows, with each row representing a different employee.

Table 2.1: Employees

Employee_Number	First_name	Last_Name	Date_of_Birth	Car_Number
10001	Axel	Washington	28-Aug-43	5
10083	Arvid	Sharma	24-Nov-54	null
10120	Jonas	Ginsberg	01-Jan-69	null
10005	Florence	Wojokowski	04-Jul-71	12
10099	Sean	Washington	21-Sep-66	null
10035	Elizabeth	Yamaguchi	24-Dec-59	null

The primary key for this table would generally be the employee number because each one is guaranteed to be different. (A number is also more efficient than a string for making comparisons.) It would also be possible to use First_Name and Last_Name because the combination of the two also identifies just one row in our sample database. Using the last name alone would not work because there are two employees with the last name of "Washington." In this particular case the first names are all different, so one could conceivably use that column as a primary key, but it is best to avoid using a column where duplicates could occur. If Elizabeth Taylor gets a job at this company and the primary key is First_Name, the RDBMS will not allow her name to be added (if it has been specified that no duplicates are permitted). Because there is already an Elizabeth in the table, adding a second one would make the primary key useless as a way of identifying just one row. Note that although using First_Name and Last_Name is a unique composite key for this example, it might not be unique in a larger database. Note also that Table 1.2 assumes that there can be only one car per employee.

`SELECT` Statements

SQL is a language designed to be used with relational databases. There is a set of basic SQL commands that is considered standard and is used by all RDBMSs. For example, all RDBMSs use the SELECT statement.

A SELECT statement, also called a query, is used to get information from a table. It specifies one or more column headings, one or more tables from which to select, and some criteria for selection. The RDBMS returns rows of the column entries that satisfy the stated requirements. A SELECT statement such as the following will fetch the first and last names of employees who have company cars:

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Car_Number IS NOT NULL
```

The result set (the set of rows that satisfy the requirement of not having null in the Car_Number column) follows. The first name and last name are printed for each row that satisfies the requirement because the SELECT statement (the first line) specifies the columns First_Name and Last_Name. The FROM clause (the second line) gives the table from which the columns will be selected.

```
FIRST_NAME               LAST_NAME
----------               -----------
Axel                     Washington
Florence                 Wojokowski
```

The following code produces a result set that includes the whole table because it asks for all of the columns in the table Employees with no restrictions (no **WHERE** clause). Note that **SELECT *** means "SELECT all columns."

```
SELECT *
FROM Employees
```

WHERE Clauses

The **WHERE** clause in a **SELECT** statement provides the criteria for selecting values. For example, in the following code fragment, values will be selected only if they occur in a row in which the column Last_Name begins with the string 'Washington'.

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Last_Name LIKE 'Washington%'
```

The keyword **LIKE** is used to compare strings, and it offers the feature that patterns containing wildcards can be used. For example, in the code fragment above, there is a percent sign (%) at the end of 'Washington', which signifies that any value containing the string 'Washington' plus zero or more additional characters will satisfy this selection criterion. So 'Washington' or 'Washingtonian' would be matches, but 'Washing' would not be. The other wildcard used in **LIKE** clauses is an underbar (_), which stands for any one character. For example,

```
WHERE Last_Name LIKE 'Ba_man'
```

would match 'Batman', 'Barman', 'Badman', 'Balman', 'Bagman', 'Bamman', and so on.

The code fragment below has a **WHERE** clause that uses the equal sign (=) to compare numbers. It selects the first and last name of the employee who is assigned car 12.

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Car_Number = 12
```

The next code fragment selects the first and last names of employees whose employee number is greater than 10005:

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Employee_Number > 10005
```

WHERE clauses can get rather elaborate, with multiple conditions and, in some DBMSs, nested conditions. This overview will not cover complicated **WHERE** clauses, but the following code fragment has a **WHERE** clause with two conditions; this query selects the first and last names of employees whose employee number is less than 10100 and who do not have a company car.

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Employee_Number < 10100 and Car_Number IS NULL
```

A special type of **WHERE** clause involves a join, which is explained in the next section.

2.1.1 Joins

A distinguishing feature of relational databases is that it is possible to get data from more than one table in what is called a join. Suppose that after retrieving the names of employees who have company cars, one wanted to find out who has which car, including the make, model, and year of car. This information is stored in another table, Cars, shown in Table 1.3.

Table 1.3. Cars

Car Number	Make	Model	Year
5	Honda	Civic DX	1996
12	Toyota	Corolla	1999

There must be one column that appears in both tables in order to relate them to each other. This column, which must be the primary key in one table, is called the foreign key in the other table. In this case, the column that appears in two tables is Car_Number, which is the primary key for the table Cars and the foreign key in the table Employees. If the 1996 Honda Civic were wrecked and deleted from the Cars table, then Car_Number 5 would also have to be removed from the Employees table in order to maintain what is called referential integrity. Otherwise, the foreign key column (Car_Number) in Employees would contain an entry that did not refer to anything in Cars. A foreign key must either be null or equal to an existing primary key value of the table to which it refers. This is different from a primary key, which may not be null. There are several null values in the Car_Number column in the table Employees because it is possible for an employee not to have a company car.

The following code asks for the first and last names of employees who have company cars and for the make, model, and year of those cars. Note that the **FROM** clause lists both Employees and

Cars because the requested data is contained in both tables. Using the table name and a dot (.) before the column name indicates which table contains the column.

```
SELECT Employees.First_Name, Employees.Last_Name, Cars.Make,
           Cars.Model, Cars.Year
FROM Employees, Cars
WHERE Employees.Car_Number = Cars.Car_Number
```

This returns a result set that will look similar to the following:

```
FIRST_NAME   LAST_NAME      MAKE        MODEL     YEAR
-----------  ------------   --------   ---------  -------
Axel         Washington     Honda       CivicDX   1996
Florence     Wojokowski     Toyota      Corolla   1999
```

2.2 Common SQL Commands

SQL commands are divided into categories, the two main ones being Data Manipulation Language (DML) commands and Data Definition Language (DDL) commands. DML commands deal with data, either retrieving it or modifying it to keep it up-to-date. DDL commands create or change tables and other database objects such as views and indexes.

A list of the more common DML commands follows:

- **`SELECT`** – used to query and display data from a database. The **`SELECT`** statement specifies which columns to include in the result set. The vast majority of the SQL commands used in applications are **`SELECT`** statements.
- **`INSERT`** – adds new rows to a table. INSERT is used to populate a newly created table or to add a new row (or rows) to an already-existing table.
- **`DELETE`** – removes a specified row or set of rows from a table
- **`UPDATE`** – changes an existing value in a column or group of columns in a table

The more common DDL commands follow:

- **`CREATE TABLE`** – creates a table with the column names the user provides. The user also needs to specify a type for the data in each column. Data types vary from one RDBMS to another, so a user might need to use metadata to establish the data types used by a particular database. **`CREATE TABLE`** is normally used less often than the data manipulation commands because a table is created only once, whereas adding or deleting rows or changing individual values generally occurs more frequently.
- **`DROP TABLE`** – deletes all rows and removes the table definition from the database. A JDBC API implementation is required to support the DROP TABLE command as specified by SQL92, Transitional Level. However, support for the **`CASCADE`** and **`RESTRICT`** options of **`DROP TABLE`** is optional. In addition, the behavior of **`DROP TABLE`** is implementation-defined when there are views or integrity constraints defined that reference the table being dropped.

- `ALTER TABLE –` adds or removes a column from a table. It also adds or drops table constraints and alters column attributes

2.3 Result Sets and Cursors

The rows that satisfy the conditions of a query are called the result set. The number of rows returned in a result set can be zero, one, or many. A user can access the data in a result set one row at a time, and a cursor provides the means to do that. A cursor can be thought of as a pointer into a file that contains the rows of the result set, and that pointer has the ability to keep track of which row is currently being accessed. A cursor allows a user to process each row of a result set from top to bottom and consequently may be used for iterative processing. Most DBMSs create a cursor automatically when a result set is generated.

Earlier JDBC API versions added new capabilities for a result set's cursor, allowing it to move both forward and backward and also allowing it to move to a specified row or to a row whose position is relative to another row.

2.4 Transactions

When one user is accessing data in a database, another user may be accessing the same data at the same time. If, for instance, the first user is updating some columns in a table at the same time the second user is selecting columns from that same table, it is possible for the second user to get partly old data and partly updated data. For this reason, DBMSs use transactions to maintain data in a consistent state (data consistency) while allowing more than one user to access a database at the same time (data concurrency).

A transaction is a set of one or more SQL statements that make up a logical unit of work. A transaction ends with either a commit or a rollback, depending on whether there are any problems with data consistency or data concurrency. The commit statement makes permanent the changes resulting from the SQL statements in the transaction, and the rollback statement undoes all changes resulting from the SQL statements in the transaction.

A lock is a mechanism that prohibits two transactions from manipulating the same data at the same time. For example, a table lock prevents a table from being dropped if there is an uncommitted transaction on that table. In some DBMSs, a table lock also locks all of the rows in a table. A row lock prevents two transactions from modifying the same row, or it prevents one transaction from selecting a row while another transaction is still modifying it.

2.5 Stored Procedures

A stored procedure is a group of SQL statements that can be called by name. In other words, it is executable code, a mini-program, that performs a particular task that can be invoked the same way one can call a function or method. Traditionally, stored procedures have been written in a DBMS-specific programming language. The latest generation of database products allows stored procedures to be written using the Java programming language and the JDBC API. Stored procedures written in the Java programming language are bytecode portable between DBMSs.

Once a stored procedure is written, it can be used and reused because a DBMS that supports stored procedures will, as its name implies, store it in the database.

The following code is an example of how to create a very simple stored procedure using the Java programming language. Note that the stored procedure is just a static Java method that contains normal JDBC code. It accepts two input parameters and uses them to change an employee's car number.

Do not worry if you do not understand the example at this point. The code example below is presented only to illustrate what a stored procedure looks like. You will learn how to write the code in this example in the tutorials that follow.

```
import java.sql.*;

public class UpdateCar {

    public static void UpdateCarNum(int carNo, int empNo)
                                                throws SQLException {
        Connection con = null;
        PreparedStatement pstmt = null;

        try {
          con = DriverManager.getConnection("jdbc:default:connection");

            pstmt = con.prepareStatement(
                        "UPDATE EMPLOYEES SET CAR_NUMBER = ? " +
                        "WHERE EMPLOYEE_NUMBER = ?");
            pstmt.setInt(1, carNo);
            pstmt.setInt(2, empNo);
            pstmt.executeUpdate();
        }
        finally {
            if (pstmt != null) pstmt.close();
        }
    }
}
```

2.6 Metadata

Databases store user data, and they also store information about the database itself. Most DBMSs have a set of system tables, which list tables in the database, column names in each table, primary keys, foreign keys, stored procedures, and so forth. Each DBMS has its own functions for getting information about table layouts and database features. JDBC provides the interface `DatabaseMetaData`, which a driver writer must implement so that its methods return information about the driver and/or DBMS for which the driver is written. For example, a large number of methods return whether or not the driver supports a particular functionality. This interface gives users and tools a standardized way to get metadata. In general, developers writing tools and drivers are the ones most likely to be concerned with metadata

Module 1: Performance Tuning on Relational Database

Unit 3: Performance Tuning on Relational Database

3.1 **Introduction**

The performance of a DBMS on commonly asked queries and typical update operations is the ultimate measure of a DB design.

A DB Architecture DBA can improve performance by adjusting some DBMS parameters (.e.g. the size of buffer pool) and by identifying performance bottlenecks and adding hardware to eliminate such bottlenecks.

After the design of the conceptual scheme by creating a collection of relations and views along with a set of integrity constraints, we must address performance goals through physical db design (physical schema).

However, performance tuning is usually necessary as user requirements evolve, that is, tuning or adjusting all aspects of a db design for good performance.

3.2 **Need for Db turning**

Since accurate, detailed workload (lists of queries, update & their frequencies) information may be hard to come by while doing the initial design of the system, therefore, tuning a db after it has been designed &deployed is important i.e. we must refine the initial design in the light of actual usage patterns to obtain the best possible performance.

3.3 **DB tuning overview**

Design process could be considered to be over once an initial conceptualschema is designed and a set of indexing and dustering decisions is made. However, any subsequent changes to the

conceptual schema or the indexes would be regarded as a tuning activities; in which many of the original assumptions about the expected workload can be replaced by observed usage patterns. It may lead to validation or cancelling of some of the initial workload specification.
For instance, the initial guesses about the size of data can be replaced with the actual statistics from the system catalogs (although this information will keep changing as the system evolve).

3.4 **Types of DB tuning**

There are 3 kinds of tuning: tuning indexes, tuning the conceptual schema, and tuning queries.

Tuning indexes: If the observed workload reveals that some queries &updates considered important in the initial workload specification are not very frequent &some new queries &updates are identified to be important; then the choice of indexes they would be reviewed in line with this new information. Thus, some of the original indexes may be dropped &new ones added.

For example:
SELECT D. mgr
FROM Employees E, Departments D
WHERE D.dname= 'Toy' AND E.dno =D.dno

A good plan here would be to use an index on dname to retrieve Departments tuples with dname ='Toy' and to use a dense index on the dno field of Employees as the inner relation, using an index-only scan.
Now suppose that queries of this form take an unexpectedly longtime to execute. We can also see the plan produced by the optimizer, but if the plan indicates that an index-only scan is not being used, but that Employees tuples are being retrieved, we have to refine/tune our initial choice of index on the dno field of Employees and to replace it with a clustered index.
Tuning the conceptual schema: This is the redesign of the relation scheme during the initial design process or later, after the system has been in use for a while, if our current choice of relation scheme does not enable us to meet our performance objective for the given workload.

Options to consider while taking the conceptual schemes are:
* We may decide to settle for 3NF design instead of a BNF design
* If there are 2 ways to decompose a given scheme into 3NF or BCNF, our choice should be guided by the workload.
* We might decide to further decompose a relation that is already in BNCF
* We might demoralize; choose to replace a collection of relations obtained by decomposition from a larger relation with the original (large) relation with identical schemes.
* We might partition a relation horizontally; which would lead to our having two relations with identical schemas.

Tuning queries and views: If query is running slower than expected, we can examine the query carefully to find the problem and some rewriting of the query, perhaps in conjunction with some index thing, can often fix the problem. When thing a query, the first thing is to verify that the system is using the plan that you expect it to use. It may be that the system is not finding best

plan for a variety of reason e.g selection condition involving mull values, arithmetic strong expressions inability to do index-only seen.

Tuning Query example:
SELECT E.dno
FROM Employee
WHERE E.hobby = ‘stamps’ OR E.age=10
If we have index on both hubby and age, we can use these indexes to retrieve the necessary tuples, but an optimize might fail to recognize this opportunity. The optimizer might view the conditions in the WHERE clause as a wholes and as not matching either index.
Suppose we rewrite the query as the union of two queries one with the clause WHERE E .age=10. Now each of these queries will be answered efficiently with the aid of the indexes on hobby and age.

Example 2: Sometime a query with GROUP BY and HAVING can be replaced by a query without these clauses, thereby eliminating a sort operation.
SELECT MIN(E.age)
FROM Employees E
GROUP BY E.dno
HAVING E.dno =102
This query is equivalent to
SELECT MIN (E.age)
FROM Employees E
WHERE E.dro=102

Examples:
Complex queries are often written in steps, using a temporary relation. We can usually rewrite such queries without the temporary relation to make then run faster. Consider the following query for computing the average salary of department managed by Robinson:
SELECT *
INTO TEMP
FROM Employee E Department D
WHERE E.dno= D E.dno AND D.mgr name = ‘Robinson’
SELECT T.dno, AVG (T.sal)
FROM Temp T
GROUP BY T.dno
This query can be rewritten as:
SELECT E.dno, AVG(E.sal)
FROM Employees E, Departments D
WHERE E.dno AND D. mgrname = ‘ Robinson’
The rewritten query does not materialize relation the intermediate relation.
Temp and is therefore likely to be faster.

Performance tuning of RDBMS aims at executing SQL Queries with the least possible response time. Although newer relational databases and faster hardware run most SQL queries with a

significantly small response time, there is always room for improvement.

There are so many relational databases and examples include, ORACLE,DB2,MS SQL SERVER,INFORMIX and so on, but they all share same design concepts under their hood and therefore we shall be discussing some techniques that are applicable to virtually all forms of RDBMS.

3.5 TECHNIQUES FOR PERFORMANCE TURNING:

Database statistics

The most important resource to any SQL optimizer is the statistics collected for different tables within the catalog. Statistics is the information about indexes and their distribution with respect to each other. Optimizer uses this information to decide the least expensive path that satisfies a query. Outdated or missing statistics information will cause the optimizer to take a less optimized path hence increasing the overall response time.

SQL commands for different database that is used to update statistics.

Oracle:	ANALYZE command or DBMS_UTILITY package
DB2:	RUNSTATS command
MS SQL Server:	UPDATE STATISTICS

Optimizers always tend to select the least expensive path \u2013 one that returns least number of rows in fastest time. Why do optimizers rely on statistics? Consider the following query that is run against our sample database to answer this question.

```
select *
from customer
where city = 'New York City'
and phone = '212-555-1212'
```

Notice that the above query contain two fields in the "WHERE" clause and there are two indexes defined, each containing one field. One very important notion to remember is that the optimizer can only use ONE index per table. Therefore, it has to make a decision as to which index to use. Since phone number should return least amount of rows, our query will run much faster if the optimizer always uses IdxPhone. However, if statistics are not updated, the optimizer does not know which index is better and may decide to choose IdxCity since 'city' field appears first in our WHERE clause. Once you update statistics the database will know more about the data distribution and will correctly choose the better index to run your query.

Avoid functions on RHS of the operator

Often developers use functions or method with their SQL queries. Consider the following example.

```
SELECT  *
from Customer
 where YEAR(AccountCreatedOn) == 2005
and  MONTH(AccountCreatedOn) = 6
```

Note that even though AccountCreatedOn has an index, the above query changes the where clause such a way that this index cannot be used anymore.

Rewriting the query in the following way will increase the performance tremendously.

```
Select *
From Customer
Where AccountCreatedOn between '6/1/2005'
   and '6/30/2005'
```

Specify optimizer hints in SELECT

Although in most cases the query optimizer will pick the appropriate index for a particular table based on statistics, sometimes it is better to specify the index name in your SELECT query. For example, consider the following

```
SELECT *
FROM customer
WITH ( Index(IdxPhone))
WHERE city = 'New York City'
      and phone = '212-555-1212'
```

Notice the additional "WITH" clause after FROM. This example is specific to MS SQL Server. Every database use different syntax for specifying this value and they are quite different from each other. Refer to your RDBMS documentation for details.

Use EXPLAIN

Most databases return the execution plan for any SELECT statement that is created by the optimizer. This plan is very useful in fine tuning SQL queries. The following table lists SQL syntax for different databases.

Oracle:	EXPLAIN PLAN FOR >Your query<
DB2:	EXPLAIN PLAN SET queryno = xxx for >Your query<
MS SQL Server:	Set SHOWPLAN_ALL ON >Your query<
Informix:	SET EXPLAIN
Sybase ASE:	Set SHOWPLAN_ALL ON >Your query<

You can also use third party tools, such as WinSQL Professional from Synametrics Technologies
to run EXPLAIN commands against databases.

Avoid foreign key constraints

Foreign keys constraints ensure data integrity at the cost of performance. Therefore, if performance is your primary goal you can push the data integrity rules to your application layer.
A good example of a database design that avoids foreign key constraints is the System tables in most databases. Every major RDBMS has a set of tables known as system tables. These tables contain meta data information about user databases. Although there are relationships among these tables, there is no foreign key relationship. This is because the client, in this case the database itself, enforces these rules.

3.6 ***USE MULTIPLE STORAGE DEVICE***

Hard disk I/O is among the slowest resource on a computer, which becomes apparent as the size of your database increase. Many databases allow users to split their database onto multiple physical hard drives. In fact, some even go a step further and allow splitting the contents of a table on multiple disks. When you use multiple physical disks, I/O operations speed up significantly since more heads fetch data in parallel

.

SELECT ONLY NEEDED DATA'S

The less data retrieved, the faster the query will run. Rather than filtering on the client, push as much filtering as possible on the server-end. This will result in less data being sent on the wire and you will see results much faster. Eliminate any obvious or computed columns. Consider the following example.

```
Select  FirstName, LastName, City
Where City  = 'New York City'
```

In the above example, you can easily eliminate the "City" column, which will always be "New York City". Although this may not seem to have a large effect, it can add up to a significant value for large result sets.

Drop indexes before loading data

Consider dropping the indexes on a table before loading a large batch of data. This makes the insert statement run faster. Once the inserts are completed, you can recreate the index again.

If you are inserting thousands of rows in an online system, use a temporary table to load data. Ensure that this temporary table does not have any index. Since moving data from one table to another is much faster than loading from an external source, you can now drop indexes on your primary table, move data from temporary to final table, and finally recreate the indexes.

3.7 CONCLUSION

The techniques highlighted above would greatly enhance the performance tuning of your relational database.

Module 2: Querying Database

Unit 1: QUERYING GRAPH STRUCTURED DATABASE

1.0 Introduction

2.0 Motivating Example

3.0 Preliminaries

4.0 Architecture

5.0 Anatomy of the Index Manager

5.1 Complete Index on Quadruples

5.2 Index Structure Candidates

5.3 Implementing a Complete Index on Quads

6.0 Indexers and Data Placement

7.0 Distributed Query Evaluation

7.1 Atomic Lookups over the Network

7.2 Join Processing

YAR2: A Federated Repository for Querying Graph Structured Data from the Web

1 Introduction

The technological underpinnings of the Web are constantly evolving. With markup and representation languages, we have witnessed an upgrade from HTML to XML, mainly in the blogosphere where early adopters embraced the XML-based RSS (Really Simple Syndication) format to exchange news items. Data encoded in XML is better structured than HTML due to stricter syntax requirements and the tagging of data elements as opposed to document elements. Although the XML web is smaller in size than the HTML web, specialised search engines make use of the structured document content. Whilst XML is appropriate in data transmission scenarios where actors agree on a fixed schema prior to document exchange, ad-hoc combination of data across seemingly unrelated domains rarely happens. Collecting data from multiple XML sources requires applications to merge data. The data merge problem is addressed by RDF, whereby, ideally, identifiers in the form of URIs are agreed-upon across many sources. In this scenario, RDF data on the Web organises into a large well-linked directed labelled graph that spans a large number of data sources.

There is an abundance of data on the Web hidden in relational databases, which represents a rich source of structured information that could automatically be published to the Web. Some weblog hosting sites have already begun exporting RDF user profiles in the Friend of a

Friend (FOAF) vocabulary. Community driven projects such as Wikipedia and Science Commons, and publicly funded projects – for example, in the cultural heritage domain – plan to make large amounts of structured information available under liberal licence models. Hence, we see the benefit of a system that allows for interactive query answering and large-scale data analysis over the aggregated Web structured-data graph. We study such a system as part of the Semantic Web Search Engine (SWSE) project. The goal of SWSE is to provide an end-to-end entity-centric system for collecting, indexing, querying, navigating, and mining graph-structured Web data.

The system will provide improved search and browsing functionality over existing web search systems; returning answers instead of links, indexing and handling entity descriptions as opposed to documents. The core of SWSE is YARS2 (Yet Another RDF Store, Version 2), a distributed system for managing large amounts of graph-structured data. Our work unifies experience from three related communities: information retrieval, databases, and distributed systems. We see our main contribution as identifying suitable well-understood techniques from traditional computer systems research, simplifying and combining these techniques to arrive at a scalable system to manage massive amounts of graph-structured data collected from the World Wide Web.

The remainder of this paper is organised as follows:

1. We describe the architecture and modus operandi of a distributed Web search and query engine operating over graph-structured data.
2. We present a general indexing framework for RDF, instantiated by a read optimised,\ compressed index structure with near-constant access times with respect to index size.
3. We investigate different data placement techniques for distributing the index structure.
4. We present methods for parallel concurrent query processing over the distributed index.
5. We provide experimental measurements of scaling up the system to billions of statements.

2 Motivating Example

In the following we describe a scenario which current search engines fail to address: to answer structured queries over a dataset combined from multiple Web sources. A well interlinked graph-structured dataset furthermore enables new types of mining applications to detect common patterns and correlations on Web scale.
The use-case scenario is to find mutual acquaintances between two people. More specifically, the query is as follows: give me a list of people known to both Tim Berners-Lee and Dave Beckett. The query can be answered using data combined from a number of different sources. Having aggregated all data from the sources, a query engine can evaluate the query over the combined graph. For our example query, Dan Brickley is one resulting answer to the question of who are mutual acquaintances of Tim and Dave?, that can only be derived by considering data integrated from a number of sources.

From the motivating example we can derive a number of requirements:

– Keyword searches. The query functionality has to provide means to determine the identifier of an entity1 which can be found via keyword based searches (such as tim berners lee).

– Joins. To follow relationships between entities we require the ability to perform lookups on the graph structure. We cater for large result sets for high level queries, which is in contrast to Web searches where typically only the first few results are relevant.
– Web data. Since we collect data from the open Web environment, we need to pre-process the data (e.g., fusing identifiers); in addition, the index structures have to be domain independent to deal with schema-less data from the Web.
– Scale. Anticipating the growth of data on the Web, a centralised repository aggregating available structured content has to scale competently. The system has to exhibit linear scale-up to keep up with fast growth in data volume. A distributed architecture is imperative to meet scale requirements. To allow for good price/benefit ratio, we deploy the system on commodity hardware through use of a shared-nothing paradigm.
– Speed. Answers to interactive queries have to be returned promptly; fast response times are a major challenge as we potentially have to carry out numerous expensive joins over data sizes that exceed the storage capacity of one machine. To achieve adequate response times over large amounts of data, the indexing has to provide constant lookup times with respect to scale.

3 Preliminaries

Before describing the architecture and implementation of our system, we provide definitions for concepts used throughout the paper.

Definition 1. (RDF Triple, RDF Node) Given a set of URI references R, a set of blank nodes B, and a set of literals L, a triple (s, p, o) € (RUB)×R×(RUBUL) is called an RDF triple.
In a triple (s, p, o), s is called subject, p predicate or property, and o object. To be able to track the provenance of a triple in the aggregated graph, we introduce the notion of context.

Definition 2. (Triple in Context) A pair (t, c) with a triple t and c € (R U B) is called a triple in context c.
Please note that we refer to a triple ((s, p, o), c) in context c as a quadruple or quad (s, p, o, c). The context of a quad denotes the URL of the data-source from hence the contained triple originated.

4. Architecture

We present the distributed architecture of SWSE, combining techniques from databases and information retrieval systems. A system orientated approach is required for graph-based data from the Web because of scale. The system architecture of a Semantic Web Search Engine requires the following components:

– Crawler. To harvest web-documents, we use Multi Crawler : a pipelined crawling architecture which is able to syntactically transform data from a variety of sources (e.g., HTML, XML) into RDF for easy integration into a Semantic Web system.
– Indexer. The Indexer provides a general framework for locally creating and managing inverted keyword indices and statement indices; we see these two index types as the fundamental building blocks of a more complex RDF index. Our framework, with combinations of keyword and statement indices, can be used to implement specialised systems for indexing RDF.

– Object Consolidator. Within RDF, URIs are used to uniquely identify entities. However, on the web, URIs may not be provided or may conflict for the same entities. We can improve the linkage of the data graph by resolving equivalent entities. For example, we can merge equivalent entities representing a particular person through having the same values for an email property;
– Index Manager. The Index Manager provides network access to the local indices, offering atomic lookup functionality over the local indices. Local indices can include keyword indices on text and statement indices such as quad indices on the graph structure, and join indices on recurring combinations of data values.
– Query Processor. The Query Processor creates and optimises the logical plan for answering both interactive browsing and structured queries.
The Query Processor then executes the plans over the network in a parallel multi-threaded fashion, accessing the interfaces provided by the local Index Managers resident on the network.
– Ranker. To score importance and relevance of results during interactive exploration, we use ReConRank . ReConRank is a links analysis technique which is used to simultaneously derive ranks of entities and data-sources.
Ranking is an important addition to search and query interfaces and is used to prioritise presentation of more pertinent results.
– User Interface. To provide user-friendly search, query and browsing over the data indexed, we provide a user interface which is the human access point to the Semantic Web Search Engine.
Users incrementally build queries to browse the data-graph – through paths of entity relationships – and retrieve information about entities.

The focus of the paper is on describing YARS2 (Yet Another RDF Store, Version 2),), the indexing and query processing functionality as illustrated in Figure 1. In the remainder of the paper, we first describe the Index Manager, next discuss the Indexer and data placement strategies, and then present the Query Processor.
Fig. 1. Parallel index construction and query processing data flow.

5. Anatomy of the Index Manager

We require index support to provide acceptable performance for evaluating queries. The indices include

- a keyword index to enable keyword lookups.
- quad indices to perform atomic lookup operations on the graph structure
- join indices to speed up queries containing certain combinations of values, or paths in the graph.

For the keyword index, we deploy Apache Lucene2, an inverted text index. The keyword index maps terms occurring in an RDF object of a triple to the subject. We implement the quad index using a generic indexing framework using (key, value) pairs distributed over a set of machines. Similarly, join indices can be deployed using the generic indexing architecture. In the following, we illustrate the indexing framework using the quad index; join indices can be deployed analogously.

5.1 Complete Index on Quadruples

The atomic lookup construct posed to our index is a quadruple pattern.

Definition 3. (Variable, Quadruple Pattern) Let V be the set of variables. A quadruple (s, p, o, c) € (RUBUV) × (RUV) × (RUBULUV) × (RUBUV) is called a quadruple pattern.

A naive index structure for RDF graph data with context would require four indices: on subject, predicate, object, and context. For a single quad pattern lookup containing more than one constant, such a naive index structure needs to execute a join over up to four indices to derive the answer. Performing joins on the quad pattern level would severely hamper performance. Instead, we implement a complete index on quads which allows for direct lookups on multiple dimensions without requiring joins. If we abstract each of the four elements of a quad pattern as being either a variable V or a constant C = RUBUL, we can determine that there are 24 = 16 different quad lookup patterns for quadruples. Naively, we can state that 16 complete quad indices are required to service all possible quad patterns; however, assuming that prefix lookups are supported by the index, all 16 patterns can be covered by six alternately ordered indices. Prefix lookups allow the execution of a lookup with a partial key; in our case an incomplete quad.

We continue by examining three candidate data structures for providing complete coverage of the quad patterns. In examining possible implementations, we must also take into account the unique data distribution inherent in RDF. The most noteworthy example of skewed distribution of RDF data elements is that of rdf : type predicate; almost all entities described in RDF are typed. Also, specific schema properties can appear regularly in the data. Without special consideration for such data skew, performance of the index would be impacted.

5.2 Index Structure Candidates

For implementing a complete index on quadruples, we consider three index structures:
B-tree, hash table, and sparse index.
– **A B-tree** index structure provides prefix lookups which would allow us to implement a complete index on quads with only six indices as justified in Section 5.1; one index can cover multiple access patterns. However, assuming a relatively large number of entries (106 −109), the logarithmic search complexity requires prohibitively many disk I/O operations (20 - 30) given that we are limited as to the portion of the B-tree we can fit into main memory.
– **Hash-tables** enable search operations in constant time; however, a hash table implementation does not allow for prefix lookups. A complete index on quads implemented using hash tables would thus require maintaining all 16 indices. The distribution of RDF data elements is inherently skewed; elements such as rdf:type would result in over-sized hash buckets. If the hash value of a key collides with such an oversized bucket, a linear scan over all entries in the hash bucket is prohibitively expensive.
– A third alternative, and the one we implement, is that of a sparse index, which is an in-memory data structure that refers to an on-disk sorted and blocked data file. The sparse index holds the first entry of each block of the data file with a pointer to the on-disk location of the respective block. To perform a lookup, we perform binary search on the sparse index in memory to determine the position of the block in the data file where the entry is located, if present. With the sparse index structure, we are guaranteed to use a minimum number of on-disk block accesses, and thus achieve constant lookup times similar to hash tables. Since the sparse index allows for prefix lookups, we can use concatenated keys for implementing the complete index structure on quads.

5.3 Implementing a Complete Index on Quads

The overall index we implement comprises of an inverted text index and six individual blocked and sorted data files containing quads in six different combinations. For the sparse indices over the data files, we only store the first two elements of the first quad of each block to save memory at the expense of more data transfers for lookups keys with more than two dimensions.

More generally, the sparse index represents a trade-off decision: by using a smaller block size and thus more sparse index entries, we can speed up the lookup performance. By using a larger block size and thus less sparse index entries, we can store more entries in the data file relative to main memory at the expense of performance. The performance cost of larger block sizes is attributable to the increase of disk I/O for reading the larger blocks.

To save disk space for the on-disk indices, we compress the individual blocks using Huffman coding. Depending on the data values and the sorting order of the index, we achieve a compression rate of _ 90 %. Although compression has a marginal impact on performance, we deem that the benefits of saved disk space for large index files outweigh the slight performance dip.

Figure 2 shows the correspondence between block size and cumulated lookup time for 100k random lookups, and also shows the impact of Huffman coding on the lookup performance; block sizes are measured pre-compression. The average lookup time for a data file with 100k entries using a 64k block size is approximately 1.1 ms for the uncompressed and 1.4 ms for the compressed data file. For 90k random lookups over a 7 GB data file with 420 million synthetically generated triples, we achieve an average seek time of 8.5 ms.

6 Indexers and Data Placement

The Indexer component handles the local creation of the keyword and sparse indices for the given data. For our specific complete quad index, we require building six distinctly ordered, sorted and compressed files from the raw data.
The following outlines the process for local index creation orchestrated by the
Indexer component:

1. Block and compress the raw data into a data file ordered in subject, predicate, object, context order (SPOC).
2. Sort the SPOC data file using a multi-way merge-sort algorithm.
3. Reorder SPOC to POCS and sort the POCS data file.
4. Complete step 3 for the other four index files.
5. Create the inverted text index from the sorted SPOC index file.

We performed an initial evaluation of the multi-way merge-sort of a file containing over 490M quads. We sorted segments of the file in memory, wrote the

Fig. 2. Effect of block size on lookup performance using uncompressed and compressed blocks.

We performed random lookups on all keys in a file containing 100k entries with varying block sizes. Results plotted on log/log scale.

Sorted quads to batch files, and then merge-sorted the resulting batch files. Depending on the size of the in-memory segments, the process took between 19 hours 40 minutes (80k statements in-memory) and 9 hours 26 minutes (320k statements in-memory).

Thus far, we have covered local index management. Since our index needs to implement a distributed architecture for scalability and we require multiple machines running local Index Managers, we need to examine appropriate data placement strategies.

We consider three partitioning methods to decide which machine(s) a given quad will be indexed on:

1. Random placement with flooding of queries to all machines
2. Placement based on a hash function with directed lookup to machines where quads are located
3. Range-based placement with directed lookups via a global data structure

We focus on the hash-based placement, which requires only a globally known hash function to decide where to locate the entry. The hash placement method can utilise established distributed hash table substrates to add replication and fail safety. For more on how to distribute triples in such a network.

We avoid complex algorithms to facilitate speed optimisation. The peer to which an index entry (e.g. SPOC, POCS) is placed is determined by:

peer(entry) = h(entry[0]) mod m
where m is the number of available Index Managers.

Hashing the first element of an index entry assumes an even distribution of values for the element which is not true for predicates. The issue of load balancing based on query forwarding in hash-distributed RDF stores has been investigated. However, a simpler solution which does not require query forwarding is to resort to random distribution where necessary (for POCS), where the index is split into even sizes, and queries are flooded to all machines in parallel.

To evaluate the indexing component, we created a univ(50000) dataset using the Lehigh University Benchmark [12], which we adapted to also produce variable-length text strings from an English dictionary in order to test Lucene.
Table 1 summarises the indices deployed for the scale-up experiments.

Table 1. Index statistics for synthetically generated dataset.

7 Distributed Query Evaluation

We implement a general-purpose query processor operating on multiple remote Index Managers to enable evaluation of queries in SPARQL format3. In this section, we

- discuss network lookup optimisations for stream-processing large result sizes and evaluate our approach with a dataset of 7 billion statements
- devise a query processing method to perform joins over the distributed Index Managers.

7.1 Atomic Lookups over the Network

Before we can perform join processing in the Query Processor, we must implement optimised methods for handling the network traffic and memory overhead involved in sending large amounts of atomic lookup requests and receiving large amounts of response data over the network, to and from the remote Index Managers.

We implement multi-threaded requests and responses between the Query Processor and the Index Managers. For example, with our flooding distribution, each machine in the network receives and processes the lookup requests in parallel.
To be able to handle large result sets, we have to be careful not to overload main memory with intermediate results that occur during the query processing and therefore we need a streaming results model where the main memory requirements of the machines are finite since results are materialised in-memory as they are being consumed.

For a quad pattern lookup, multiple remote Index Managers are probed in parallel using multiple threads. The threaded connections to the Index Managers output results into a coordinating blocking queue with fixed capacity. The multiple threads synchronise on the queue and pause output if the queue capacity is reached.

Iterators that return sets instead of tuples to increase performance have been described in as row blocking. We measure the impact of row blocking via an index scan query over 2, 4, 8, and 16 Index Managers. Each index manager provides access to a over 7 GB data file with 420 million synthetically generated triples, which amounts to a total capacity of roughly 7 billion statements. To be able to test keyword performance, we changed the string values in the Lehigh benchmark to include keywords randomly selected from a dictionary.

Figure 3 shows the impact of varying row blocking buffer size on the network throughput. As can be seen, throughput remains constant despite increasing the number of Index Managers servicing the index scan query. From this we can conclude that a bottleneck exists in the machine consuming results.

Fig.3. Throughput for index scans with varying row blocking sizes.

7.2 Join Processing

We begin our discussion of join processing by introducing the notions of variable bindings, join conditions, and join evaluation and continue by detailing our method of servicing queries which contain joins.

Definition 4. (Variable bindings) A variable binding is a function from the set of variables V to the set of URI references R, blank nodes B, or literals L.

Definition 5. (Join Condition) Given multiple quad patterns in a query, a join condition exists between two quad patterns Qj and Qk iff there exists one variable v € V, v € Qj , v € Qk. Joins are commutative. Variable v is termed the join variable.

In our query processing system, a query may consist either of one quad pattern (an atomic lookup) or may consist of multiple quad patterns where each pattern satisfies the join condition with at least one other pattern.
For joins we use a method called index nested loops join. Multiple join operations can run concurrently in individual threads, with queues as coordination data structures for data exchange between the operators. Figure 4 illustrates the parallel execution of joins across remote Index Managers coordinated by the main thread M. Queues are represented as stack of boxes. Thread S represents a lookup operations of the first quad pattern in a query. The lookup is flooded to n Index Managers via threads S1...n. The alternative would be to perform a directed lookup via the hash function. Intermediate results are passed to the join thread J, which in turn floods the lookups to n Index Managers via threads J1...n. Threads J1...n write final join evaluations to a blocking queue, which is accessed by the main thread M.

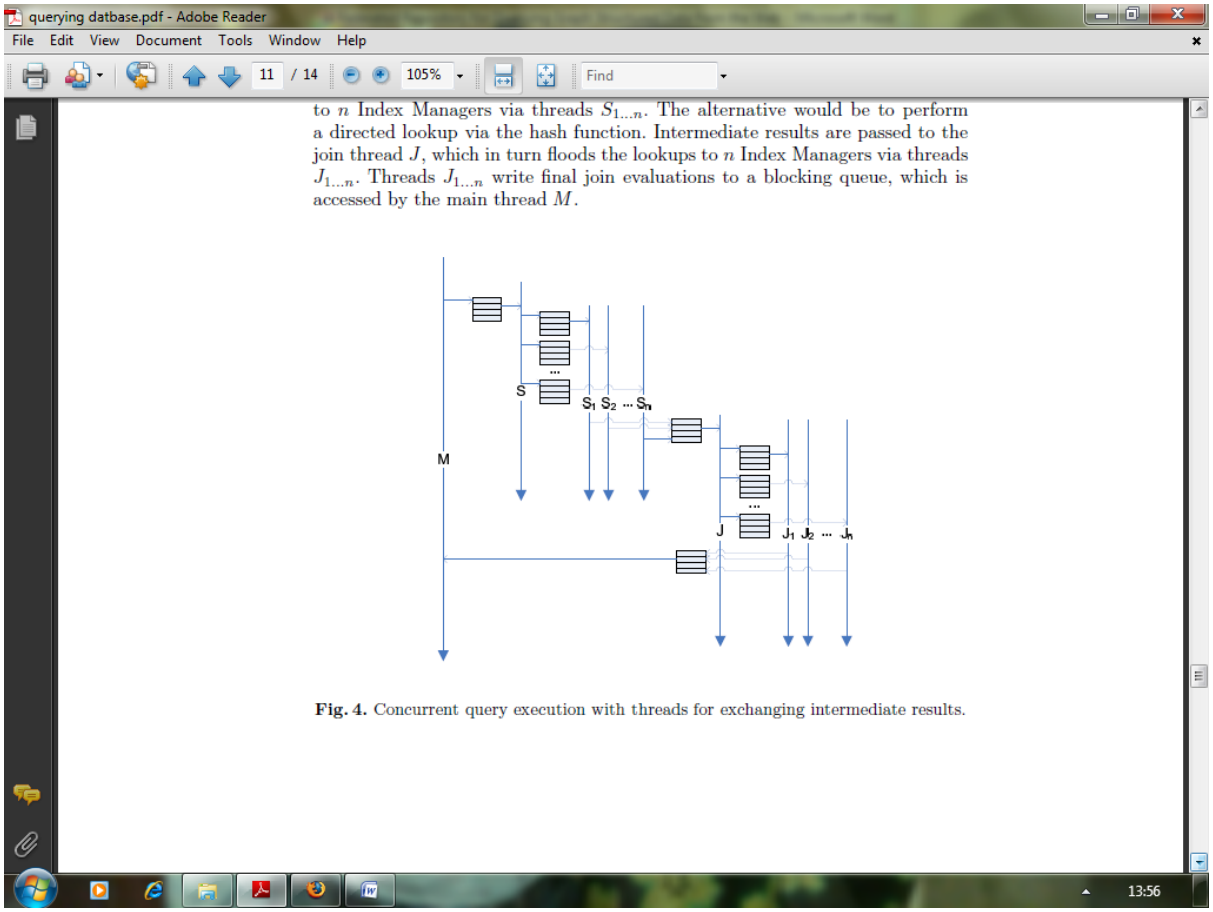


Fig. 4. Concurrent query execution with threads for exchanging intermediate results.

Unit 2: **QUERY STREAM DATABASE**

INTRODUCTION

Data streams are common in many recent applications, e.g. Stock quotes, e-commerce data, system logs, network traffic management, etc. Compared with traditional databases, streaming databases pose new challenges for query processing due to the streaming nature of data which constantly changes over time. Index structures have been effectively employed in traditional databases to improve the query performance. Index building time is not of particular interest in static databases because it can easily be amortized with the performance gains in the query time. However, because of the dynamic nature, index building time in streaming databases should be negligibly small in order to be successfully used in continuous query processing.

WHAT IS A QUERY

A database query is the vehicle for instructing a DBMS to update or retrieve specific data to/from the physically stored medium. The actual updating and retrieval of data is performed through various "low-level" operations. Examples of such operations for a relational DBMS can be relational algebra operations such as project, join, select Cartesian product, etc. While the DBMS is designed to process these low-level operations efficiently.

We define streaming database as a collection of multiple data streams, each of which arrives sequentially and describes an underlying signal. For example, the data feeds from a sensor network form a streaming database. The dimensionality of each data stream is always increasing in this case. Hence, theoretically the amount of data stored, if stored at all, in a streaming database tends to be infinite. This leaves us with a challenge of trying to get accurate query results from a huge database with time constraints. Moreover, in most cases users would expect fast response time for queries. This makes it necessary to develop an effective index structure for streaming database with very efficient update cost, so that query results can be obtained in a tolerable amount of time.

THE QUERY PROCESSOR

There are three phases that a query passes through during the DBMS' processing of that query:

1. Parsing and translation
2. Optimization
3. Evaluation

Most queries submitted to a DBMS are in a high-level language such as SQL.
During the parsing and translation stage, the human readable form of the query is translated into forms usable by the DBMS. These can be in the forms of a relational algebra expression, query tree and query graph. Consider the following SQL query:

Select *make*
From ***vehicles***
Where ***make*** **="**ford**"**

This can be translated into either of the following relational algebra expressions:
$\sigma_{\text{make-"ford"}}(\pi_{\text{make}}(\text{vehicles}))$
$\pi_{\text{make}}(\sigma_{\text{make-"ford"}(}\text{vehicles}))$
It can be represented as a query tree and also as a query graph
After parsing and translation into a relational algebra expression, the query is then transformed into a form, usually a query tree or graph, that can be handled by the optimization engine. The optimization engine then performs various analyses on the query data, generating a number of valid evaluation plans. From there, it determines the most appropriate evaluation plan to execute

PARSING AND TRANSLATING THE QUERY

The first step in processing a query submitted to a DBMS is to convert the query into a form usable by the query processing engine. High-level query languages such as SQL represent a query as a string, or sequence, of characters. Certain sequences of characters represent various types of tokens such as keywords, operators, operands, literal strings, etc. Like all languages, there are rules (syntax and grammar) that govern how the tokens can be combined into understandable (i.e. valid) statements. The primary job of the parser is to extract the tokens from the raw string of characters and translate them into the corresponding internal data elements (i.e. relational algebra operations and operands) and structures (i.e. query tree, query graph). The last job of the parser is to verify the validity and syntax of the original query string.

OPTIMIZING THE QUERY

In this stage, the query processor applies rules to the internal data structures of the query to transform these structures into equivalent, but more efficient representations. The rules can be based upon mathematical models of the relational algebra expression and tree (heuristics), upon cost estimates of different algorithms applied to operations or upon the semantics within the query and the relations it involves. Selecting the proper rules to apply, when to apply them and how they are applied is the function of the query optimization engine.

EVALUATING THE QUERY

The final step in processing a query is the evaluation phase. The best evaluation plan candidate generated by the optimization engine is selected and then executed.
Note that there can exist multiple methods of executing a query. Besides processing a query in a simple sequential manner, some of a query's individual

operations can be processed in parallel—either as independent processes or as interdependent pipelines of processes or threads. Regardless of the method chosen,
the actual results should be same.

QUERY METRICS: COST

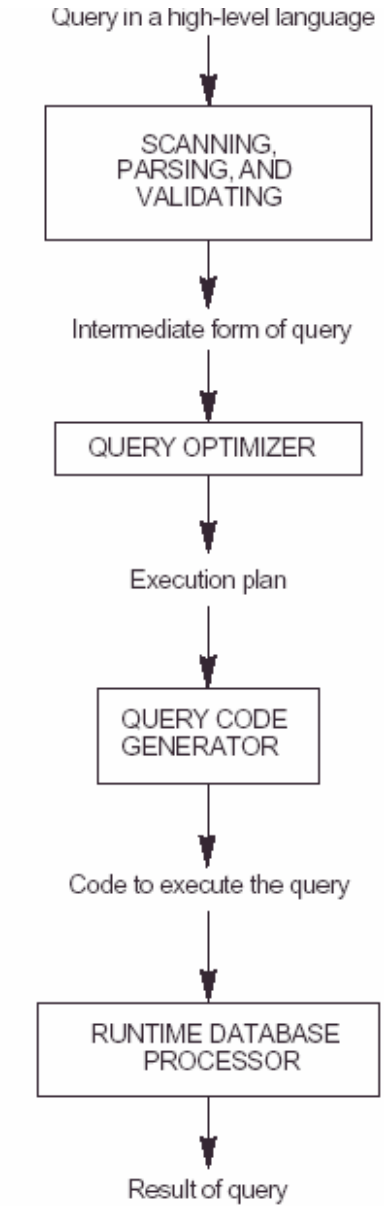
The execution time of a query depends on the resources needed to perform the needed operations: disk accesses, CPU cycles, RAM and, in the case of parallel
and distributed systems , thread and process communication (which will not be considered in this paper). Since data transfer to/from disks is substantially slower than memory-based transfers, the disk accesses usually represent an overwhelming
majority of the total cost—particularly for very large databases that cannot be pre-loaded into memory. With today's computers, the CPU cost also can be insignificant compared to disk access for many operations. The cost to access a disk is usually measured in terms of the number of blocks transferred from and to a disk, which will be the unit of measure referred to in the remainder of this paper.

QUERY PROCESSING ARCHITECHTURE

- Query node represents a single block query, and are generated code
- All query node live in a run time system, and follow an API.
- LOW LEVEL QUERIES
  - Limited set of query nodes (selection/projection, aggregation)
  - Tight constraints on resource usage
- HIGH LEVEL QUERIES
  - Much wider variety of operators
  - Use operator templates, specialized with generated functions
  - Accepts tuple callback routes tuples through operators in the query node

BASIC STEPS IN QUERY PROCESSING

1) The scanning, parsing, and validating module produces an internal representation of the query.
2) The query optimizer module devises an execution plan which is the execution strategy to retrieve the result of the query from the database files. A query typically has many possible execution strategies differing in performance, and the process of choosing a *reasonably efficient* one is known as query optimization.
3) The code generator generates the code to execute the plan.
4) The runtime database processor runs the generated code to produce
   the query result.



BASIC ALGORITHMS FOR EXECUTING RELATIONAL QUERY OPERATIONS

• An RDBMS must include one or more alternative algorithms that implement each relational algebra operation (SELECT, JOIN,…) and, in many cases, that implement each combination of theses operations.
• Each algorithm may apply only to particular storage structures and access paths.
• Only execution strategies that can be implemented by the RDBMS algorithms and that apply to the particular query and particular database design can be considered by the query optimization module.

ALGORITHM FOR IMPLEMENTING SELECT OPERATIONS
These algorithms depend on the file having specific access paths and may apply only to certain types of selection conditions.
Many search methods can be used for simple selection: S1 through S6;
S1: Linear Search (brute force) –full scan in Oracle's terminology
– Retrieves every record in the file, and test whether its attribute values satisfy the selection condition: an expensive approach.

S2: **Binary Search**
– If the selection condition involves an equality comparison on a key attribute on which the file is ordered.
S3: **Using a Primary Index (hash key)**
– An equality comparison on a key attribute with a primary index (or hash key).
– This condition retrieves a single record (at most).
S4: Using a primary index to retrieve multiple records
– Comparison condition is >, >=, <, or <= on a key field with a primary index
– Use the index to find the record satisfying the corresponding equality condition, then retrieve all subsequent records in the (ordered) file.
S5: Using a clustering index to retrieve multiple records
– If the selection condition involves equality comparisons on a non-key attribute with a clustering index.
S6: Using a secondary (B-tree) index on an equality comparison
– The method can be used to retrieve a single record if the indexing field is a key or to retrieve multiple records if the indexing field is not a key.

ALGORITHM FOR IMPLEMENTING JOIN OPERATIONS
J1: Nested-loop join (brute force)
– For each record (outer loop) retrieve every record from the (inner loop) and test whether the two records satisfy the join condition.
J2: Single-loop join (using an access structure to retrieve the matching records)
– If an index (or hash key) exists for one of the two join attributes , retrieve each record one at a time (single loop), and then use the access structure to retrieve directly all matching records that satisfy that satisfy the join condition
J3: Sort-merge join
– If the records of R and S are physically sorted (ordered) by value of the join attributes A and B, respectively, we can implement the join in the most efficient way.
– Both files are scanned concurrently in order of the join attributes, matching the records that have the same values for A and B.
– If the files are not sorted, they may be sorted first by using external sorting.
– Pairs of file blocks are copied into memory buffers in order and records of each file are scanned only once each for matching with the other file if A & B are key attributes.
– The method is slightly modified in case where A and B are not key attributes.
J4: Hash-join
– The records of files are both hashed to the same hash file using the same hashing function on the join attributes as hash keys.

REFERENCES

[1] Henk Ernst Blok, Djoerd Hiemstra and Sunil Choenni, Franciska de Jong, Henk M. Blanken and Peter M.G. Apers. Predicting the cost-quality trade-off for information retrieval queries: Facilitating database design and query optimization. *Proceedings of the tenth international conference on Information and knowledge management*, October 2001, Pages 207-214.

[2] D. Calvanese, G. De Giacomo, M. Lenzerini and M. Y. Vardi. Reasoning on Regular Path Queries. *ACM SIGMOD Record*, Vol. 32, No. 4, December 2003.

[3] Ramez Elmasri and Shamkant B. Navathe. Fundamentals of Database Systems, second edition. Addison-Wesley Publishing Company, 1994.

[4] Donald Kossmann and Konrad Stocker. Iterative Dynamic Programming: A new Class of Query Optimization Algorithms. *ACM Transactions on Database Systems*, Vol. 25, No. 1, March 2000, Pages 43-82.

[5] Reza Sadri, Carlo Zaniolo, Amir Zarkesh and Jafar Adibi. Expressing and Optimizing Sequence Queries in Database Systems. *ACM Transactions on Database Systems*, Vol. 29, Issue 2, June 2004, Pages 282-318.

INDEXING HIGH DIMENSIONAL DATA

Many advanced technologies have been developed to store and record large quantities of data continuously in the recent years. In many cases, the data may contain errors or may be only partially complete. For example, sensor networks typically create large amounts of uncertain data sets. In other cases, the data points may correspond to objects which are only vaguely specified, and are therefore considered uncertain in their representation.

Furthermore, the dimensions which contribute most to the distance between a pair of records are also likely to have the greatest uncertainty. Therefore, the effects of high dimensionality are magnified by the uncertainty, and the contrast in distance function computations is lost.

The problem of indexing has been studied extensively in the literature both for the case of deterministic data and for the case of uncertain data. However these techniques do not deal with some of the unique challenges in the similarity indexing of high dimensional or uncertain data.

These unique challenges are as follows:

- Similarity functions need to be carefully designed for the high dimensional and uncertain case in order to maintain contrast in similarity calculations. Furthermore, the distance function needs to be sensitive to the use of an index.
- In most cases, the similarity or range queries are only performed on a small subset of the dimensions of high dimensional data. For example, in many applications, we are likely to perform a range query only over 3 to 4 dimensions of a 100-dimensional data set. Such queries cannot be processed with the use of traditional index structures, which are designed for full dimensional queries. The queries which can be resolved with the use of our index structure are as follows:
- Determine the nearest neighbor to a given target record in conjunction with an effective distance function.

INDEXING HIGH DIMENSIONAL DATA

There are some unique challenges, in the similarity indexing of high dimensional or uncertain data, which cannot be dealt with or handled by the deterministic data indexing techniques.
These unique challenges are:

- Similarity functions need to be carefully designed for the high dimensional and uncertain case in order to maintain contrast in similarity calculations. Also, the distance function needs to be sensitive to the use of an index.

- In most cases, the similarity or range queries are only performed on a small subset of the dimensions of high dimensional data. i.e. performing a range query only over 3 to 4 dimensions of a 100-dimensional data set.

The queries which can be resolved with the use of high-dimensional index structure are:

- Query to determine the nearest neighbor to a given target record in conjunction with an effective distance function.

- Query to determine the nearest the neighbor to the target T by counting the expected number of dimensions for which the points lie within user-specified threshold distances tI……td.

- Query to determine the points which lie in R(s) with probability greater than d , in a given subset of dimensions S, and a set of ranges R(s) defined on the set S.

UniGrid: (Uncertain Inverted GRID Structure): it is an efficient indexing structure for uncertain data and high-dimensional data.

UniGrid structure has a two-level inverted partitions for data there is an inverted list of record identifiers separately for each identifiers for all points whose mean value and uncertainty lie within certain pre-specified ranges.

UniGrid can be used for querying processes involving similarity queries and range queries.

This is shown in the algorithms below:
Algorithm similarity Query Database: D, Target Point: {Y, h(.)},
Thresholds: t1…….td);
{ (z^I ……z^d) represent midpoints of corresponding probability density function spans}.

Begin
Determine the span for the uncertainty function h (.) along the different dimensions and denote by O^I…….O^d;
Determine all the inverted lists for which the range
$[Z^k-O^k-t_k, Z^k+O^k-t_k]$ intersects with the corresponding inverted list range [Ls (k,r,t), us (k,r,t)];

Compute similarity values for the different record identifiers in these inverted lists by aggregating the computation of the probabilistic function $h(y^k,xi^k,t_k)$ over different dimensions; report largest possible similarity value and record identifier;

Similarity Search Processing

Algorithm Range Query (Database: D, Dimensions: qI……..v), Rangers:[a_1, b_1]…… [a_v, b_v];
Threshold: d);
Begin
Determine all inverted lists along the first dimension for which [L_s (1,r,t), us (1,r,t)] intersects with the range (a_1, b_1] compute probability of intersection of corresponding data points in inverted lists with range (a1, b1) using the uncertainty function of corresponding data points and denote list by L_1; Remove data points list L_1 with probability less than d ; for dim=2 to v Do
Begin
Determine all inverted lists along the first dimension for which [L_s (dim,r,t), us (dim,r,t)] intersects with the range [a_{dim}, b_{dim}];
Compute probability of intersection of corresponding data points in inverted lists with range [a_{dim}, b_{dim}]
Using the uncertainty function of corresponding data points and denote list by L_2;
Multiply the probabilities of list L1 with the probabilities in list L2;
{It is assumed that the absent elements have probability of zero}
Remove data point in list L_1 with probability less than d ;
End
Return (L1);

MANAGING UNCERTAIN IN DATABASE

An uncertain database is a database in which objects do not have precise positions i.e. an object can be anywhere in a circular uncertainty region.

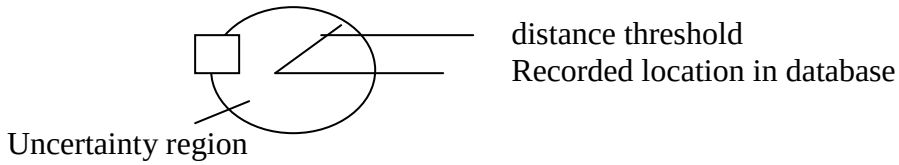


Figure I: An uncertain object.

In an uncertain database, an object O is associated with a multi-dimensional probability density function (pdf), which describes the likelihood that O appears at each position in the data base. This occurs mostly in numerous applications that manage "dynamic attributes" with continuously changing values.

Examples of Applications in which uncertain data management techniques are relevant are:

i) Location-based services: In which a moving client informs a server about its coordinates, if its distance from the previously reported location exceeds a certain threshold. Here, there are no clients' precise positions.

2) A meteorology system: This monitors the temperatures, humidity, UV indexes in a large number of regions. Since the readings are taken by sensors in local areas and transmitted to a central database periodically (e.g. at interval of 30mins), the db content may not exactly reflect the current atmospheric status i.e. the actual temperature in the region might have changed since it was last measured.

3) In privacy-preserving data mining applications.

4) Demographic data sets.

CAUSES OF UNCERTAINTY IN DATABASE

- Limitations of the underlying: The output of sensor networks is often uncertain, because of the noise in the sensor inputs, errors in wireless transmission or delay in data update.

- Data randomness/ Availability of only partially aggregated data sets: Demographic datasets is often uncertain since the data sets are actually a probability distribution.

- Imputation and survey of data: In privacy-preserving data mining applications; the data is perturbed in order to preserve the sensitivity of attribute values. Also data attributes are constructed using survey, imputation or forecasting. However, these attributes are usually uncertain.

INFORMATION RETRIEVAL FROM UNCERTAIN DATA

It is meaningless, retrieving information directly from uncertain data, since the resultant information does not have any quality guarantees.

For example;

Consider the query: “find the clients currently in the downtown area”.

Returning simply the objects whose last updates satisfy the query is inadequate, because many objects might have entered or left the search region since they contacted the server last time.

To avoid this problem, the “precise” values need to be estimated using a probability density function (pdf).

For example:

If the location of a moving client O is considered uniformly distributed in its uncertainty region Ur, the object pdf can be represented as pdf(x)=1/AREA (ur) if the parameter x (any point in the 2 dimensional data space)belongs to ur; or O otherwise.

Thus, the appearance probability that O lies in a given region rq (e.g, the rectangle in Figure I) equals:

$$\int_{rq \cap ur} pdf(x)dx = \frac{AREA(rq \cap ur)}{AREA\ (Ur)}$$

Where the integral area rq ∩ur is the intersection between rq and ur.

Therefore, an “uncertain object” is a multidimensional point whose location can appear at any point x in the data space, subject to a probability density function pdf(x).

i.e. uncertain object O is associated with:

(i) a probability density function o.pdf(x), where x is an arbitrary d-dimensional point, and

(ii) a d-dimensional uncertainty region o.ur

UNCERTAIN DATA MANAGEMENT APPLICATIONS

These include applications such as query processing, Online Analytical Processing, Selectivity estimation, indexing and join processing.

QUERYING PROCESS OF UNCERTAIN DATA

The incorporation of probabilistic information has considerable effects on the correctness and computability of the query plan in uncertain Data.

A given query over an uncertain database may require computation or aggregation over a large number of possibilities. In some cases, the query may be nested, which greatly increase the complexity of the computation.

There are two semantic approaches to uncertain Data querying:

(a) Intensional Semantics: It typically models the uncertain database in term of an event model and use tree-like structures of inferences on these event combinations. i.e. it enumerates all the possibilities over which the query may be evaluated and subsequently aggregated.

(b) Extensional Semantics: It attempts to design a plan which can approximate these queries without having to enumerate the entire tree of inferences.

INDEXING UNCERTAIN DATA

The lands of queries which can be resolved with the use of index structures are:

- Range queries: This aims at finding all the objects in a given range. Since the objects are uncertain, their exact positions cannot be known, hence their membership in the range also cannot be known deterministically. Therefore, a probability value is associated for each object to belong to a range. All objects whose probability of membership lies above a certain threshold are retained.

- Nearest neighbor queries: It attempts to determine the objects with the least expected nearest neighbor distance to the target.

- Aggregate queries: It aims at determining the aggregate statistics from queries such as the sum or the max. Here, the interplay of different objects has to be accounted for.

TECHNIQUES INVOLVE IN PROCESSING NEAREST NEIGHBOR QUERIES

- Projection: The uncertain region of each moving object is computed based on the uncertain model used.

- Pruning: Some of the objects will be punned without having to go through the expensive process of computing their nearest neighbor probabilities.

- Bounding: Extension of the pruning to some portions of uncertainty regions which cannot be completely pruned.

- Evaluation: Calculate for each object, the probability that it is indeed the nearest neighbor to the target O.

JOIN PROCESSING ON UNCERTAIN DATA

There are 2 types of Join:

(1) Probabilistic join: Here, it is assumed that each item is associated with a range of possible values and a probability density function, which qualifies the behavior of the data over that range.

(2) Similarity join: The join is performed based on the distance between the two items.

Range Query Processing
Querying Streaming Database
Data stream systems are mostly transient data, continuous queries sytems.
Types of streams

(1) **Raw streams**: stream tuples are injected into the system by an external data source e.g. stock ticker, sensor data, network interface.

(2) **Derived streams**: streams are defined by a query expressions that yields a stream.

(3) **Archived streams**: it allows historical and real time stream content to be combined in a single database object.

Examples of Streaming Database Queries

(1) **Task**: Every second, return the total volume of trades in the previous second.

SELECT sum (volume) AS volume,

Advance_agg (qtime) AS windowtime

FROM trade {VISIBLE '1second' ADVANCE '1second'}

(2) **Task**: Every 5seconds, return the volume-adjusted price of MSFT for the last 1minutes of trades.

SELECT: sum (price*volume)/sum(volume) AS vwap,

Sum(volume) AS volume,

Advance-agg(qtime) AS windowtime

FROM trades {VISIBLE '1minute' ADVANCE '5' seconds'}

WHERE Symbol = 'MSFT'

Composing streams

- The tuples in a stream can be viewed as a series of events e.g. "The temperature in the room is 20^0", 25^0, 30^0.
- The output of a continuous query is another series of events, typically higher – level or more complex e.g.

  "The room is on fire".

Therefore, streams can be composed in various ways:

- Stream views
- Macrosemantics
- Derived streams
- Subqueries
- Active tables

A derived stream: is a database object defined by a persistent continuous query and it is always active.
Active tables: is a table with an associated continuous query.
It has 2 modes of operation (active table modes of operation):
Append: New stream tuples appended/added to table at each window
Replace: at each new window, truncate previous table contents.

DSMS

There are some Data Stream Management systems (DSMS) under 2 different categories:

(1) Open source DSMS:

(a) Esper: DSMS engine written in Tawa

(b) Telegraph CQ: postgresQL's SQL dialect, plus stream-oriented extensions.

(2) Proprietary DSMS:

(a) Stream Base

(b) Coral8

(c) Apama

Differences between Databases and Data Streams are:

Issues	**Database system**	**Data stream systems**
Model	Persistent relations	Transient relations
Relation	Tuple set/bag	Tuple sequence
Data update	Modifications	Appends
Query	Transient	Persistent
Query Answer	Exact	Approximate
Query Evaluation	Arbitrary	One pass
Query Plan	Fixed	Adaptive

Information Retrieval in Relational Database
(JDBC/ODBC,MS Access)
JDBC
With the introduction of the Java developer kit (JDK) 1.1.4, several mechanisms for accessing persistent data was built into Java. There are basically 3 different approaches; JDBC, JSQL and serialiazable objects.

JDBC: it is known as the Java Database Connectivity toolkit used to access database from Java, it consists of two levels. A low-level JDBC-driver, database vendors are expected to deliver this, which is the software responsible for establishing a connection to a specific DBMS. The other part is high-level interface which is a sort of an application interface; the model is an API where communication with the DBMS is utilized via some high-level function calls.

JDBC is meant for accessing SQL databases from an object Oriented Language, so the application programmer still needs to know SQL, since after the initialization of the communication with the DBMS, the way of accessing and updating data is done via SQL statements.

How to use JDBC
To use JDBC; the process entails the following?
* Fetching the right classes
* Loading the driver
* The actual connection and JDBC URL
* Creating statements
* Processing the result

Fetching the right classes:
Before we can use the classes in the JDBC API, we have to import these classes by a statement, that says "Import Java.sql.*.". This tells the Java complier to include a Java Classes and thereby all methods in the classified Java.sql.

Loading the driver:
In order to communicate with a DBMS, we also have to load some driver that perform the actual communication between the general JDBC API and the low-level code in the Java-driver. To do that, we have to use the Driver Manager class, the loading of the driver can be done in three different ways:
*By the command line, using the method class for name
*Just do it! New ibm.sql.DB2driver();
The last one is a quick and dirty approach, because there is no guarantee that the driver will be loaded and thereby properly initialized. Explicit load is the most correct way, and the specified driver will be loaded and from the CLASSPATH.

The Actual Connection and the JDBC URL:
The actual datasource is specified via an extended URL, and the structure of it is as follows:
Jdbc: (**subprotocol**): (**subname**)
Subprotocol is a data source and subname is typical a database, an example could be
Jdbc: db2: sample, operating on a DB2 database called sample.
The URL also has a more extended version, which is more like a network version of the JDBC URL – like file and http in a web browser.
Jdbc: (subprotocol)://hostname:port/(subname)

In order to achieve a connection, we need the Drive manager and connect class. An example of a connection to a DB2 sample database on humulus could be done with the following code:
String url= “jdbc:db2://humulus.daimi.aau.dk:4242/sample”; connection con;
Con = Driver Manager.getconnection (url);

Creating Statements:
The next step is to initialize an object in the statement class, in order to be able to execute a query. An example could be, by assuming that we have table customers in the sample database.
Statement stmt=con. Create statement ( );
Result Set rs=stmt. Execute Query (“Select from customers”);

Processing the Results:
The final task is to use the result of the query and often the result will consist of several tuples that satisfy the query.