www.DBTechNet.org

DBTech EXT Index Design and Performance Labs (IDPLabs)

With the support of the EC LLP Transversal programme of the European Union

Disclaimers

Learning Objectives:

- understanding the need and basics of indexing technologies in the mainstream DBMS
- understanding basics of index design and management in the mainstream DBMS systems

Prerequisites:

We expect that participant of the labs is familiar with SQL basics and knows how to

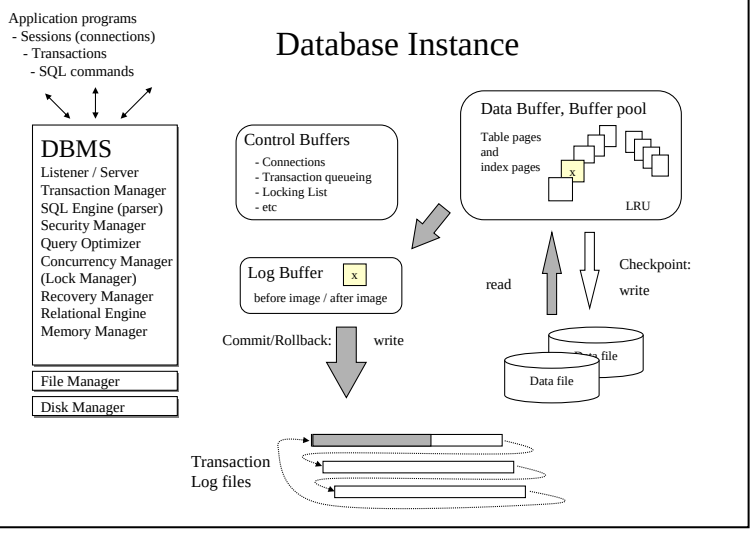
- start the database server in the selected DBMS environment
- use SQL commands by the tools of this selected DBMS

Contents

Part I - Introduction to Concepts

Databases provide the reliable storage for the persistent data of applications. Concepts of DBMS system, database instance, and database have no globally accepted standard definition, but for this tutorial we adopt the definitions for these concepts from database administrator's (DBA) point of view to the mainstream DBMS products: DB2, SQL Server, and Oracle, the "big three". A DBMS software product can be installed and used as a database server, or more precisely one or more configurable database server instances.

Figure 1. Database Server Instance



Note: In these tutorial versions of DBTechNet, with DB2 we mean DB2 for LUW (Linux, Unix or Windows) and not the mainframe editions of DB2, and all our DB2 examples have been tested using the free DB2 Express-C edition, which has proved to be an excellent tool for self-studying purposes.

Database Server Instance

A database server **instance** is built by installing a DBMS software and configuring a **named** operational set of DBMS processes which take care of various DBMS services and various **memory caches**, such as data buffer i.e. **buffer pool**, log buffers and various control buffers. The configuration is usually controlled in some control files. The instance can be started (processes instantiated) as service and stopped (shutdown). It can be configured to start automatically or manually. An instance manages one or more **databases** which have data files and transaction log files of their own, but share the buffer pool of the instance. In the following, we may use the term DBMS also for the database server instance. The database server instance provides services for the reliably storing and retrieving of data.

Database data files, pages, and transaction log

A **database** is a collection of object structures (tablespaces, tables, indexes, etc) and data which are managed as a "consistent whole". The contents of the database are stored on one or more **data files** on discs and these files are managed as **file groups** called **tablespaces**. A table or index on a table is created in a tablespace. This means that the pages of the created object will be stored in file pages of some files of that tablespace. The data files are identified internally by file numbers given by the instance, and managed

as sequence of **pages** (also called as blocks) having blocksize of 4, 8, 16, .. KB and identified accordingly by page numbers in the file.

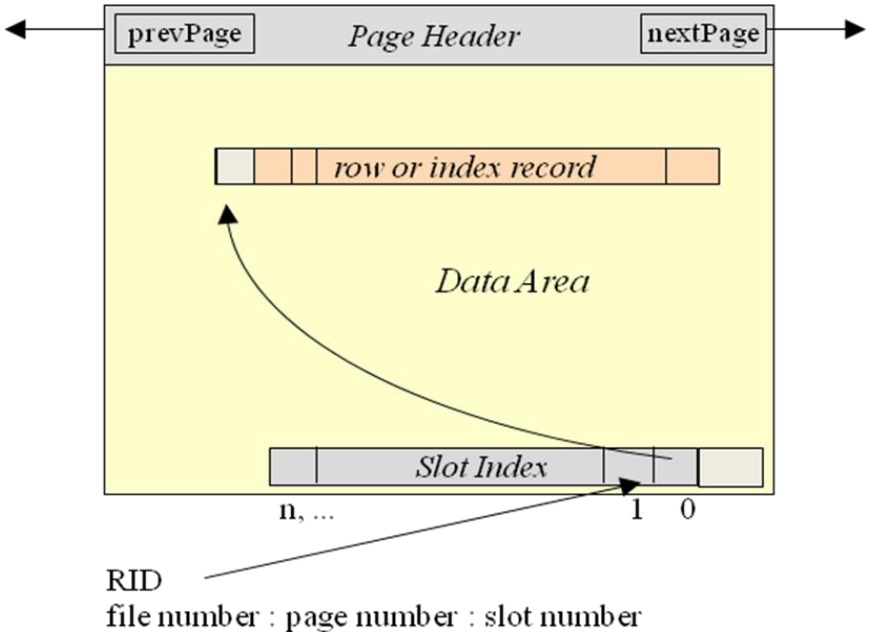


Figure 2. Typical page format

The typical page format of a table or index is presented in Figure 2. Every page has the following 3 parts:

- page header containing various control data used by the DBMS,
- data area for variable length records for storing rows or index entries of the object structure
- slot index (slot directory) containing offset addresses of the records on the data area.

Typically a page is used for rows of same table only, and the table is indicated in a field of the page header. Rows are stored on the data area records, which contain also some control information about the row, such as column lengths and offsets on the record. Row addressing is based on the indirect address **RID** (also called as ROWID, or tuple id TID) which is built from file number, page number, and slot number. In case a modified row has grown in size so that it no longer fits in the original page, the RID address remains the same but the record is split into parts which are stored on pages where there is enough room to accommodate, and the parts are chained together to preserve its original content and its sequence.

The pages of a table (or an index) are stored in double chained lists in the set of files of the tablespace into which the table (or index) was created. Read and write operations between the data files and the buffer pool occur page at a time, but for a faster sequential access in case of **pre-fetching**, a set of 8 or more pages in the chain are stored adjacent to each other, and the set is called as an **extent**. Also, for fast finding of pages having room for new records, the database maintains chains of **free list** pages.

For more detailed information of the page header fields and the record structures, we refer to textbooks and DBMS product manuals.

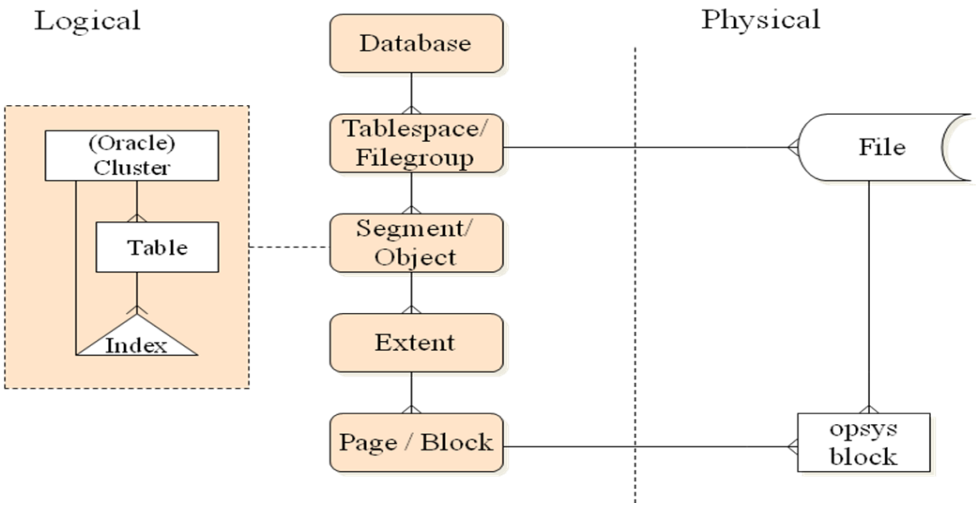


Figure 3 Relationships of storage concepts

Figure 3 presents a simplified map of storage concepts. With **objects** in the figure we mean tables and indexes, whereas Oracle calls these **segments**. Typically an extent is used for a single object only, but SQL Server starts object storages sharing the first extents with multiple objects. A table page is typically used for records of a single table, but in case of Oracle's **cluster segment**, pages can be shared between multiple tables, parent and its child tables. In this paper, we are mainly interested in database pages of tables and indexes in general.

For performance reasons, the needed index and data pages are first fetched into page frames in a **buffer pool** of the instance, allocated in the main memory of the server computer. The fetched pages remain in the buffer pool as long as there is room available, and so if a page is needed again, no disc I/O is spent for retrieving the page. This is the main performance benefit and the key for scalability of multi-user databases.

Index Technologies

Separate index structures have been used to accelerate the retrieval of data from large files already before the era of RDBMS systems, and this fairly mature technology has been adapted in all mainstream RDBMS systems, although not covered in the ANSI/ISO SQL standard, which tries to be independent of the physical implementations of relational databases. However, X/Open Group [11] of software vendors has defined X/Open SQL standard more close to the implementations and defines the concept as follows:

> "An **index** can be thought of as a list of pointers to the rows of a table, ordered based on the values of one or more specified columns of the table. Existence of an index may enhance performance by obviating certain sort operations or by reducing the scanning of the table that is necessary to build a result set."

X/Open SQL has been extended by SPIRIT 3 SQL of Service Providers' Integrated Requirements for Information Technology (SPIRIT).

The basic idea of an index, as a retrieval accelerator of rows from a table, is to copy those column values of every row in the table, which are used for the retrieval, as the **index key** value corresponding to the row, to an **index record**, and copy the **row address** (RID) on the same record. The index records are sorted according to the index key values, and organized on **index pages**, quite similar to table pages, starting with a single index page called as the **root page**, and following the recursive rule:

- when a new record does not fit on the page, the page will be **split** on two pages, and the new record is stored in its place in the sort order. Based on the last key values on these two index pages, index records with pointers to both of these pages are arranged on the index page (a new page if it did not exist before) on the next higher level as presented in Figure 4. The built tree structure is balanced, i.e. reorganized so that all the **leaf pages** of the index tree are on the same level, and on the same path distance from the root page.

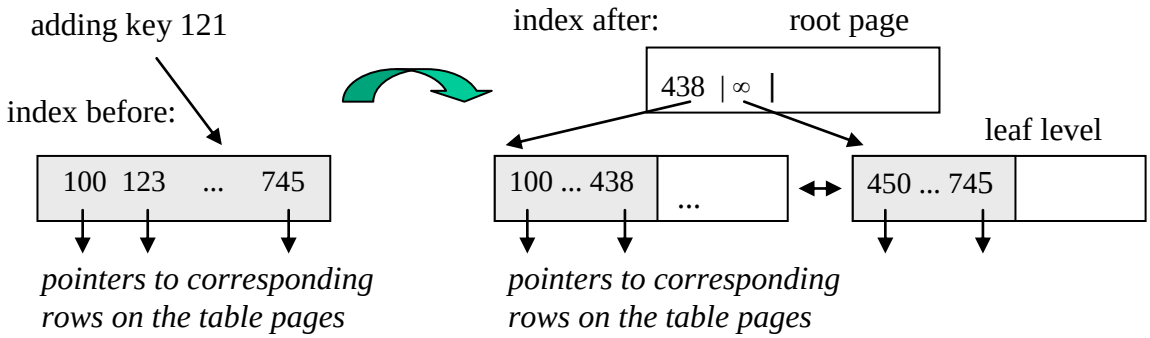


Figure 4. The first splitting in the index tree after the first page (root page) overflows

The **balanced index tree** is called a **B-tree**. There are also B-tree variations, depending on the form of the last index record on every page above the leaf level, such as **B+-tree**, which is general in the mainstream DBMS systems, and **B*-tree**. We refer to textbooks on details of these formats.

The X/Open SQL syntax for CREATE INDEX is following¨

```
CREATE [UNIQUE] INDEX [schema-name.]index-name
ON base-table-name (unqualified-column-name [ASC | DESC]
[, unqualified-column-name [ASC | DESC]]...)
```

The table and the columns referenced must exist when the index is created. The explicit or implicit schema-name of the index has to be the same schema with the base-table for which the index is created. Thus the index-names in a schema must be unique. The UNIQUE clause of the CREATE INDEX defines that the key values in the index have to be unique. This does not imply that columns of the key values cannot have NULL values, but it means that a NULL value is considered a value in index, and there cannot be duplicates with NULL values in the same key column. However, a table constraint, such as PRIMARY KEY or UNIQUE defined for the same column set will imply that no NULL values are allowed. As default, the column values of the index keys are sorted in ascending (ASC) order in the index but can be also defined to be sorted in descending order (DESC).

All mainstream RDBMS systems will generate **automatically** UNIQUE indexes for PRIMARY KEY and UNIQUE constraints with the same name as the constraint name but for FOREIGN KEYs, the index has to be created manually.

In case a table has UNIQUE indexes, whenever a row is inserted or updated, the uniqueness is first checked against all these indexes before proceeding to maintain the actual table page.

Some obsolete "rules of thumb"

The following rules of thumb have lived and been copied from textbooks to textbooks for some decades, forgetting that current the computer architecture is totally different from the architectures we had in 1980

- There should not be more than 3-5 indexes per table
- An index should include max 6 columns
- Only the root page of an index can be expected to stay in the buffer pool
- Very volatile columns should not be included in indexes.

Whenever a row in the table is inserted, updated, or deleted, it will be automatically updated in all indexes which include some of the affected columns. This will generate extra workload, and earlier it was considered such a serious performance problem that a general recommendation was that there should not be more than maximum 3 indexes per table. A typical page size in the 80's was just 2 KB, but as the modern DBMS systems are using bigger page sizes, the indexes using page size of 8 KB or more typically have 3-4 levels only even for large tables, and of those often used indexes the root page and all intermediate pages may remain in the buffer pool. The buffer pools today can be huge compared with what we had in the 80's, so that tables of a typical database may, in practice, lay in the buffer pool.

These trends, much faster computers, and more advanced DBMS technologies have outdated these "rules of thumb" raising those technical limits in the current mainstream DBMS systems. The rules of thumb may still be considered as warnings which are **worth to be verified in the environment** to be used, since it always "depends on the case". Modern PCs may be extremely powerful compared with the servers of the past, but compete in totally different series compared with really powerful servers of today. However, the following still apply

- variable length data types are not recommended as index columns, since some DBMS systems expand them to the defined maximum length for the column
- floating point data types of course do not qualify as data types in unique key columns
- LOB data types, such as CLOB, BLOB, XML, etc must not be included in SQL indexes.

Note: All the mainstream DBMS systems with which we are working in these DBTechNet labs, have an XML implementation of their own, and all have special **XML indexes** for accelerating retrieval and updates of the parts of the stored XML documents. This is a topic in our tutorial "XML, SQL/XML and the Big Three" of the "XML and Databases" lab.

Optimizer and some basic Access Methods provided by Indexes

An index of a table can accelerate data retrieval from that single table, but applications don't explicitly use indexes. Instead of that, the **access methods** to contents of a table is selected for the **access plan** of application's SQL query by the query **optimizer** of the DBMS from the following basic methods described in Figure 5:

- **Unique Matching Scan** means that for a row's existence test or fetching by an accurate index value, a direct path using fetch of just a single index page on every page level of the index sorts out the address of the requested row, for example

```
SELECT LName, FName
FROM   Cust
WHERE  CNo = '00012345' ;
```

- Range Scan will find the start of a key range like Matching Scan, and can proceed reading index records on the leaf level pages finding all row addresses of the key range values up to the upper limit of the key range provided that the index is sorted according to the order of the key range, for example

```
SELECT LName, FName
FROM   Cust
WHERE  CNo BETWEEN '00012300' AND '00012345' ;
```

  or a range of values starting with the given string pattern, for example in the following query provided that an index would be created for the column "`phone`" of the table "`Cust`"

```
SELECT LName, FName
FROM   Cust
WHERE  phone LIKE '00358999%'
```

  or a range defined for example by a value of the first index column in case we have created a **multi-column index** (also called as **combound index**) such as

```
CREATE INDEX ixm_Cust ON Cust (city, CNo);
```

  and using the query

```
SELECT CNo, LName, FName
FROM   Cust
WHERE  city = 'Luton'
```

  Here the index key consists of the concatenated values of these columns, and the index records on the leaf level are sorted in the order of the concatenated key, so the **order of the index columns counts**.

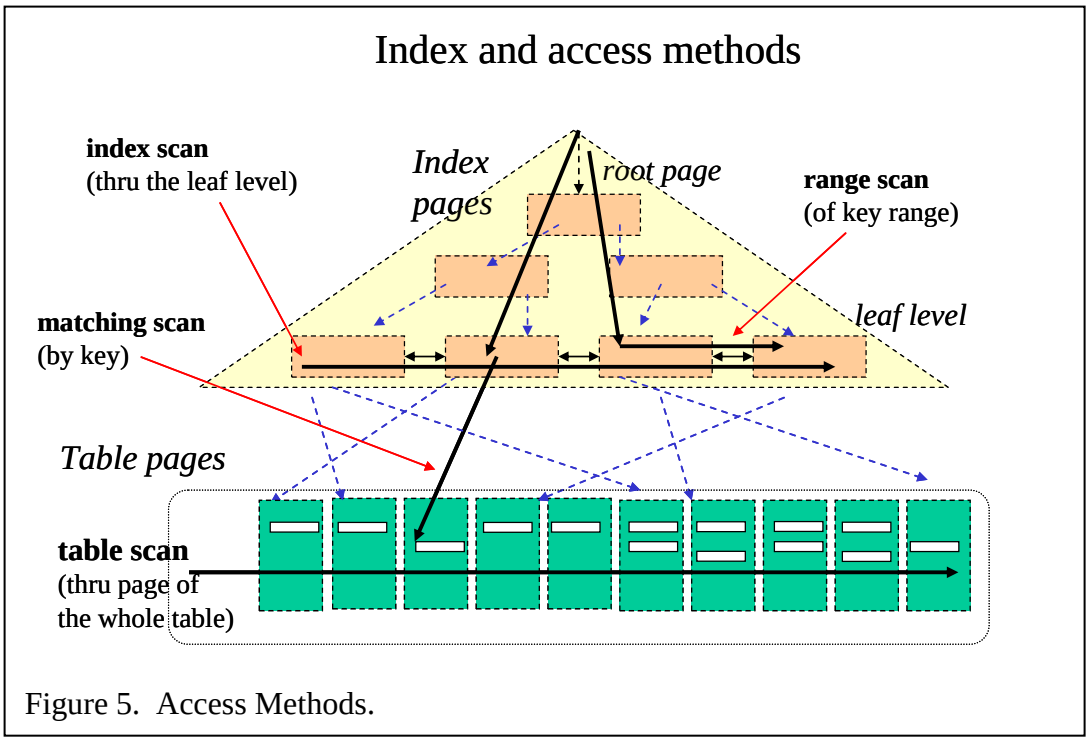


Figure 5. Access Methods.

The portion of selected rows (by the search expression predicates in the WHERE clause) from the total number of rows in the table is called **Filter Factor** (FF). For small FF values, the smaller the value is when using the index, the less disc I/O we may need for accessing the final table pages and the better the index is for the query, since disc I/O is by far the most expensive factor in terms of performance. However, for FF value over some percents, a **full table scan** can be faster, the trade-off point depending on many things, for example page size.

Now consider that we had created the index

```
CREATE INDEX ixm_Cust
    ON Cust (city, LName, FName, CNo);
```

If we enter the query

```
SELECT CNo, LName, FName
FROM   Cust
WHERE  city = 'Luton'
ORDER BY LName, FName;
```

then the search condition selects the range of keys based on the value of the first column in the key. If we compare columns in the index and columns used in the query, we find that the index covers the column needs of the query and the result set can be built based on the **index only** without accessing the table pages at all, so for this query the index is said to be a **covering index**. Further more, in this case also the **sort phase** of the result set rows is **eliminated** since we retrieve the data contents in the order defined by the ORDER BY clause. According to the Systematic Index Design methodology presented by Tapio Lahdenmäki [5], this index is a "Three-Star Index" for our query.

- An **Index Scan** browses thru all index records on the leaf level.

  If the index in our previous example is created using the column order

  ```
  CREATE INDEX ixm_Cust
      ON Cust (LName, FName, city, CNo);
  ```

  then our search cannot benefit of any range of the index keys, and we need to read all index records on the leaf level pages. Even in this case, the whole result set of our previous query can be built based on the contents of index only.

  However, if not all columns used in our query appear in the index , for example

  ```
  SELECT CNo, LName, FName
  FROM   Cust
  WHERE  city = 'Big City' AND sex = 'M'
  ORDER BY LName, FName;
  ```

  then every row pointed by the selected index records would be read by separate page read (in case these don't fit in the buffer pool after the first reading). If the filter factor of the predicate

  ```
  city = 'Big City'
  ```

  is large, then the **performance** of the query due to extra disc I/Os would be very poor, and it would be better if the optimizer would select full table scan instead of the index scan. We can find out the access plan generated by the optimizer, as every mainstream DBMS system can be asked to present (i.e. "explain") the access plan, and we can try to fix the plan
  - modifying the SQL statement, or
  - including some optimizer hints available in the SQL implementation, or
  - creating a new index which serves the query better.

- A **Full Table Scan** will browse thru all rows of the table. In terms of disc I/Os, this may not be so bad as it sounds. Part of the pages might already appear in the buffer pool. Also, instead of reading page by page from the disc, the DBMS may use **asynchronous pre-fetching** of sequence of the following pages, which eliminates the effect of disc I/Os from average random read of about 10 ms to about 40 MB/s, according to Tapio Lahdenmäki [5], which estimate would mean about 0.1-0.2 ms per page.

Clustering

A range scan or ORDER BY in case of a multi-row result set can benefit if the rows in table are in the same sort order as the index. There are also index structures, which organize the storage order of the table rows, such as

- **Clustering index** which tries to keep rows on the same table pages corresponding to the neighboring index records on the leaf level of the index,. Examples: CLUSTER index of DB2
- **Clustered index** which actually contains the whole table rows on the leaf level of the index. Examples: CLUSTERED index of SQL Server and Index organized table (IOT) of Oracle
- Oracle's **CLUSTER segment** with its index, which can store rows of both parent and child tables on the same page, according to the index used for primary key index of the parent and foreign key index of the child table. If we apply this to a single table only, the index of the cluster should act like a clustering index.

Of course, there can be only one clustering/clustered index per table, but as a rule of thumb, it is said that "there also should be one" such index.

Fat and Semi-fat Indexes

Above we presented the concept of covering index which eliminates the fetching of the actual rows from table pages. In our example, all the columns were included in the built index key. However, some systems such as DB2 and SQL Server, provide a possibility to include extra columns in the index outside the index key using the **INCLUDE clause**. For example, columns which are poor in terms of filtering, such as gender, can still be included in the index records to avoid fetching of the actual rows

```
CREATE INDEX ixm_Cust ON Cust (LName, FName, city, CNo)
INCLUDE (sex);
```

We call these kind of covering indexes **fat indexes**. In a fat index, the **key part can be unique**, but the included columns are not counted as part of the key.

Some DBMS systems, for example Oracle, don't support the INCLUDE clause. In that case we can still add extra columns at the end of the key, creating kind of **semi-fat index**, but not if the index was defined as UNIQUE.

Indexes and Constraints

Modern RDBMS systems create automatically unique indexes for PRIMARY KEY and UNIQUE constraints. However, some functionalities are not available for these automatic indexes, and in that case we first create the table without these constraints, then create the index, and finally alter the table adding the required

constraint. The DBMS then finds the existing index which might serve the added constraint, and prompts if we accept the use of the index for the constraint.

Unique indexes are used by DBMS to control uniqueness of inserted rows. Primary key index of parent table is used also as lookup index serving reference integrity control on INSERT of child table rows and on update of foreign key values of child rows.

For foreign keys some DBMS systems such as Informix, Solid, and Interbase create the index automatically, but users of the big three DBMS need to create the foreign key indexes manually. Foreign key indexes are important for accelerating performance of JOINs, and also reference integrity control of DELETE and UPDATE rules on checking if the parent row to be deleted or the primary key to be changed has child rows the child table (with the foreign key referencing the parent table).

Managing indexes

To create an index to a table, the user has to be the owner of the table or DBA or have CREATE INDEX privilege granted on the table. The index will usually be created in the same schema with the table and it needs to have a unique name in the schema. It is also possible create an index of a table in other schema than the schema of the table and using the same name as an index in the schema of the table.. In some DBMS, for example SQL Server, the index name needs to be unique only among the indexes of the same table.

Indexes can be created dynamically "online" before we have data in the table or when we already have data in the table. The maintenance of the existing indexes of a table can slow down a bulk-loading of data from external sources to the table, so it might be faster to first load the data, and then create the indexes. This should be verified by experimenting!

The previous content of the table should not violate the requirements of the index to be created, or we will "have a problem", for example in case of SQL Server, and the integrity has to be checked immediately before allowing the table to be used in production.

While creating or rebuilding an index on a table with existing rows, splitting the index pages might occur frequently and slow down the performance. To minimize splitting after the index is created, we can define how full the leaf level pages will be filled during the building process by setting the FILLFACTOR or PCTFREE parameters of the index to, for example, 50 %.

To eliminate the contention on concurrent disc head accesses, the storage of indexes of large tables should be allocated on different discs than the table. This is possible only by creating the index in a different tablespace, the files which are on different discs.

> *Note: This is not possible when experimenting with virtual computers since **our virtual discs are just files on the host computer**, we cannot experiment on this tuning possibility in our labs.*

Current cost-based optimizers need **statistics** on the tables and indexes. So, from time to time we need to collect those statistics, preferably using automatic scripts. Both tables and indexes can get **fragmented** (i.e. the records have got too scattered on disc) and to improve the performance, the fragmented object needs to be **reorganized**. In case of alternated non-clustered/clustering indexes, this can be done by dropping and **recreating** the index. In case of clustering index, the order of rows in table pages gets easily fragmented, and the usual solution to this is to drop the index, export the rows sorting them in the order of the clustering

key, then reload the rows into the table and recreate the clustering index. Some tools can make this automatically for us.

Creating an index on an existing data can be surprisingly fast, so it can be feasible solution to create a temporary index just for some query run, and drop the index after the run is ready.

Planning Additional Indexes

Even if we can have more than 3 indexes per table, the indexes need disc space and the maintenance of index records slows down the performance, especially during checkpoints.

Since we already should have indexes on primary keys, unique keys, and foreign keys, we should plan carefully the order of multicolumn keys of these, and what additional indexes we will need. For the additional indexes, we need to know what kind of transactions and queries we will need in our application and how frequently.

Before creating an index, we should have some estimate on how much disc space we need for the index. A very rough estimate in bytes can be calculated as follows

3 * number of records * (total length of columns + length of a RID)

Quick Upper-Bound Estimate (QUBE)

Even before our database has been implemented, we can calculate elapsed time estimates for planned queries based on planned indexes, and so evaluate if we should modify these. An estimation method, presented by Tapio Lahdenmäki [5] , is Quick Upper-Bound Estimate (QUBE) developed in IBM Finland.

QUBE gives us the worst case estimate of Local Response Time (LRT) of the query as total elapsed time on the local server without considering network traffic or waiting times of locks in multiuser / multiprocessing environment. The formula is following

LRT = TR * 10 ms + TS * 0.01 ms + NF * 0.1 ms

where

TR is number of Random Touches
TS is number of Sequential Touches
NF is Number of FETCH calls

Here a touch means reading of an index record or a table record (a row). Accessing a single record or the first record of adjacent records of a range slice of records, is called a **random touch**, whereas accessing a following record in a slice is called a **sequential touch**. The access times of a touch in the formula are based on averages used by Tapio Lahdenmäki in 2004. For more detailed explanations and assumptions we refer to the book [5]. The new flash technology is changing this radically faster. However, at the time we are writing this tutorial, these estimates are reasonable good. We should also bear in mind that since we typically run our virtual labs on workstations (having all our virtual discs implemented as files on a single physical disc) and we cannot eliminate effects of the disc cache of the host system.

Methodology for Systematic Index Design

Tapio Lahdenmäki [5] presents concepts of *matching* columns and *screening* columns for defining systematic design of indexes for a given SQL query, using the following example. Let's consider index iX in Figure n created for table T and the following query

```
SELECT A, B, C, D
WHERE A = :a
AND B > :b
AND C = :c
```

with *sargable* (simple enough for optimizer) predicates, where :a, :b, and :c stand for parameter values (given in host variables a, b, and c) an.

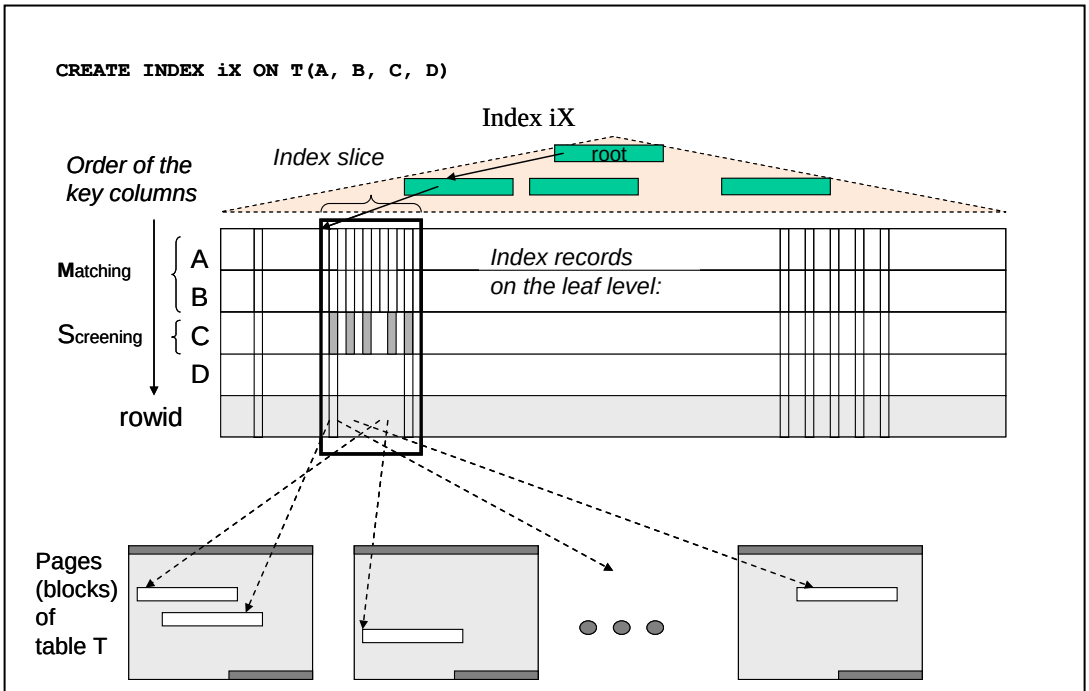


Figure n. Multi-column index iX on table T

The leading columns of the index which are used to limit the index slice of the query are called **matching columns**, so in our example A and B are the matching columns. Due to the form of predicate of B, the predicate of column C does not limit any more the index slice, so it is not considered a matching column. However, the predicate of column C may limit the need of reading actual rows in the base table T. These columns which are used to limit number of actual fetches from table pages referred by index slice records are called **screening columns**. Column D is not used for filtering purposes, but since the query does not refer to any other columns of table T, the index iX is called the covering index for the query.

If we now add the following clause to the query

ORDER BY B, C, D

then the index iX eliminates also the sort operation of the result set of the query.

Based on these concepts we can define a **simplified methodology to find / create an efficient index** for a given query Q accessing a single table T

steps:

<<1>> If there already exists some index of table T which contains all sargable columns used in the WHERE clause of the query as the leading columns of the index (i.e. as matching columns), then use this index as ix and go to step <<3>>, else
CREATE INDEX ix ON T(<list of sargable columns in the WHERE clause>).

<<2>> Calculate CUBE or run the query.
If the elapsed time is satisfactory, then go to <<end>>, else proceed to step <<3>>.

<<3>> If there already exists some index of table T which contains all sargable columns as the leading columns of the index and the index is covered index for query Q, then use this index as ix and proceed to step <<4>>, else INCLUDE all those columns referred in query Q which are not yet in the index making index a covering index for query Q
(Note: Oracle does not support the INCLUDE clause, so just add these columns as trailing columns in the index, except in case the index is UNIQUE index for some purpose)

<<4>> Calculate CUBE or run the query.
If the elapsed time is satisfactory, then go to <<end>>, else proceed to step <<5>>.

<<5>> If the query has ORDER BY clause and these columns are not matching columns, then organize the non-matching columns of the index in the order defined by the ORDER BY clause.

<<end>>

Optimizer and indexes

When a database with tables and indexes has been implemented, we have tools for more precise estimates of elapsed times and details of the execution of the queries..

The processing of a dynamic (ad hoc) SQL Query consists of the following phases

1. parsing of the SQL syntax
2. verifying of the used structures based on metadata in the system tables
3. verifying the access privileges based on metadata in the system tables
4. rewriting the query for optimization (so called query flattering)
5. optimizing the application plan
6. generating the executable code of the plan
7. execution of the application plan.

For the optimization, the Optimizer part of the DBMS looks, depending on the type of the query, for available indexes for the used tables and the statistics of these indexes and tables, calculates a set of alternative plans, and based on the estimated disc I/O times and CPU times selects the cheapest alternative as the optimized application plan.

The optimizers of DBMS systems are improving version by version, but they may be blind to some possibilities of using indexes, for example they typically don't fix data type differences. For example, in case where the primary key column CNo of table Cust is of type CHAR(8) having the corresponding unique index, an optimizer may not yet at the optimizing phase cast automatically the numeric literal `12345` to character string in

```
SELECT LName, FName
FROM   Cust
WHERE  CNo = 12345 ;
```

and does not use the index of the CNo primary key, but generates a full table scan for the plan.

All mainstream DBMS systems provide some **Explain Plan** means for us to see the generated application plan, and we can assess if the optimizer has selected the proper index accessing. We can see either the estimated plan without execution or the actually used execution plan.

For sorting out the plan of Oracle, we need to first create a special PLAN_TABLE, after which we can verify the access plan of the previous example by following

```
SQL> EXPLAIN PLAN FOR SELECT * FROM Cust WHERE CNo = 12345;
SQL> select plan_table_output
2 from table(dbms_xplan.display('plan_table',null,'serial'));
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------
| Id | Operation         | Name | Rows | Bytes | Cost |
------------------------------------------------------------------------
|  0 | SELECT STATEMENT  |      |    1 |   115 | 5681 |
|* 1 | TABLE ACCESS FULL | CUST |    1 |   115 | 5681 |
------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------
1 - filter(TO_NUMBER("CUST"."CNO")=12345)
Note: cpu costing is off
14 rows selected.
```

From this report we see that our literal argument 12345 is not of compatible data type with the CNO column. For filtering rows the DBMS first needs to cast the column value in the table to compatible data type, and thus the search predicate "`CNo = 12345`" is no more simple enough for the optimizer for using the index. We can fix this by modifying our SQL query into the form of compatible data types as

```
SELECT LName, FName
FROM   Cust
WHERE  CNo = '00012345'
```

and using this **simple enough predicate** (the term used by Lahdenmäki [5]) the optimizer will be able to filter an index slice for query. This kind of simple enough predicate is generally called "sargable", a made-up term which means that the predicate is cap***able*** to be used as simple ***s***earch ***arg***ument. If a column in the predicate needs to be casted or is used as an argument in some expression, then the predicate is called non-sargable. Sometimes a non-sargable predicate is used in purpose to avoid the use of index, for example, by adding a zero to a numeric column or concatenating an empty string to a character value, such as

```
CNo || '' = '00012345'
```

In Figure 6 we see SQL Server's estimated execution plans for a batch run of the above queries. For the first query, the DBMS has to generate a data type conversion for the search condition and for the table of million customers optimizer decides to apply **clustered index scan**, which, in this case, is a full table scan, since CNo is the primary key with clustered index, and the table pages are stored on the leaf pages of the

index. The second query uses **clustered index seek,** which we have called matching index scan, and its part of the total workload of the batch is practically zero compared with the execution plan of the first query.

An example of execution plans generated and displayed by DB2 is presented in Appendix 1, and another execution plan example of Oracle is presented in Appendix 2.

If the optimizer has not selected a proper index usage in the plan, then depending on the DBMS we can include some **optimizer hints** in the query source for the optimizer and test if this helps, or we can modify our SQL query like we did above.

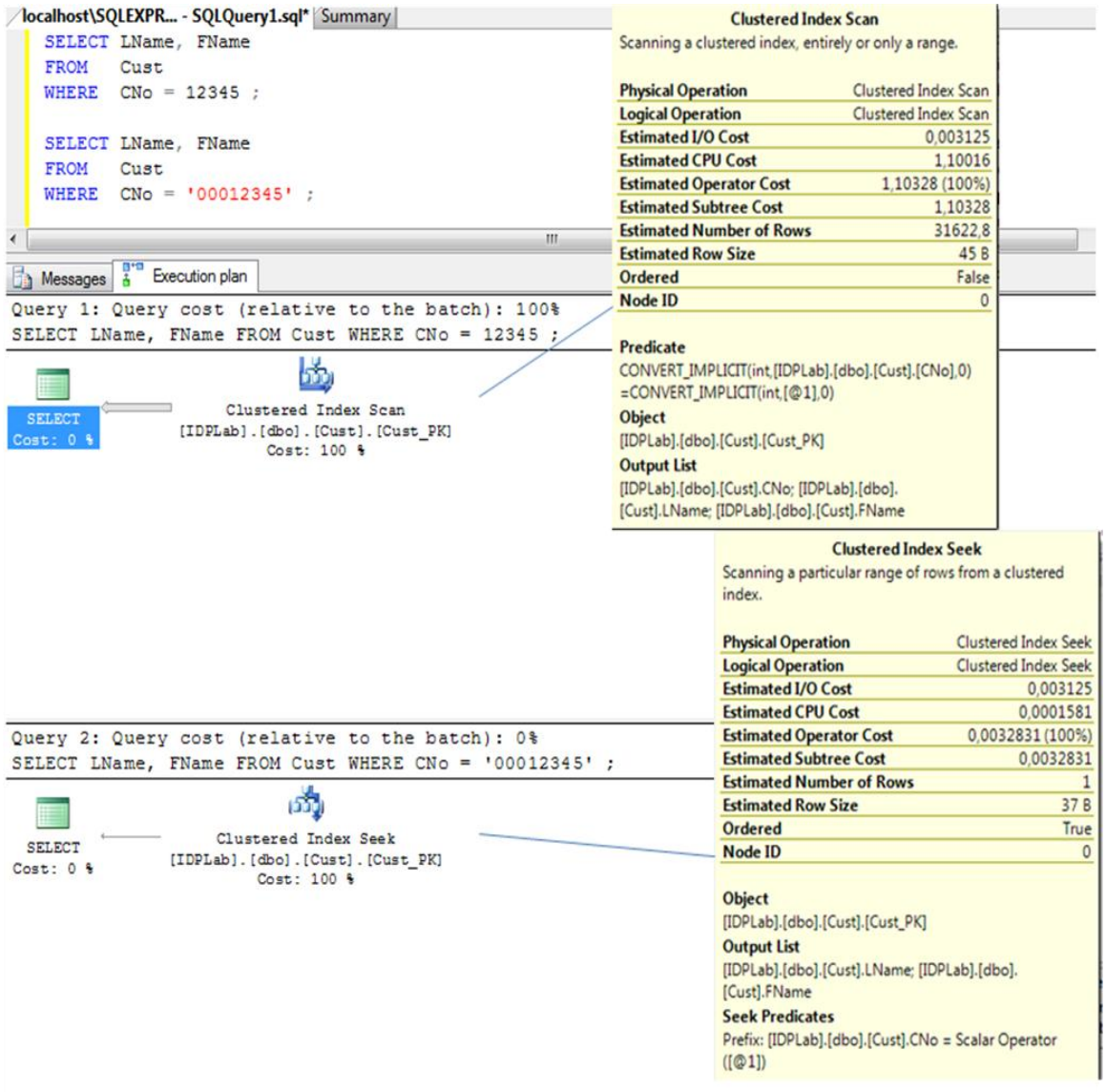


Figure 6 Execution plan estimates of the example batch by SQL Server

As part of the generated plan, the optimizer also decides on **locking requests** to be used at the execution phase, which also affects the performance, but this is not shown in the Explain Plan output. We cover the topics of locking in more detail in our concurrency control labs.

Tools and wizards

Modern DBMS systems include **tools** for

- collecting statistics of tables and key distributions in indexes to be used by the optimizer
- collecting statistics of the production queries and analyzing resource usage peaks ('nail reporting')
- analyzing fragmentation of tables and indexes
- reorganizing / rebuilding tables and indexes

and also **wizards** which based on collected production statistics suggest new indexes to improve the performance of production runs.

Advanced topics

XML Indexes

XML is now a native data type in SQL standard and the mainstream DBMS systems. Due to the different nature of XML documents, which may be up to 2 GB each, special XML indexes have been invented to accelerate the retrieval and maintenance of the parts of these documents. This is out of scope of this paper, and we cover these topics in our tutorial on "XML and Databases".

Hash indexes and Bitmap indexes

Some DBMS systems provide means for using hash structures as indexes, or static bitmap indexes for tables. Bitmap indexes are more typical in Data Warehouse databases, which is out scope of this tutorial. However, both hashing and temporary bitmap indexes can be found in generated application plans.

Indexes on Views

Some modern DBMS systems provide also means indexes on views and derived columns. This is out of scope of this paper.

Review Exercises:

(assuming that you are using, for example, DB2, SQL Server, or Oracle as the DBMS)

Note that you may not find answers of some of these questions in this tutorial

Source of the following questions: Mullins [6]

- What is the best performance tuning technique a DBA can use to improve database performance?
- What is the benefit of clustering data?
- What are the causes of database and index disorganization?
- How can file extents degrade database performance?
- What performance advantages can be gained by partitioning a table?
- What is the benefit of allocating tablespaces of tables and indexes on separate disk devices?

- Describe the impact of using the LIKE operation with a wild-card character at the beginning of the value.
- Under what circumstances is a non-matching index scan performed?
-

Part II – Index Technologies of the Big Three

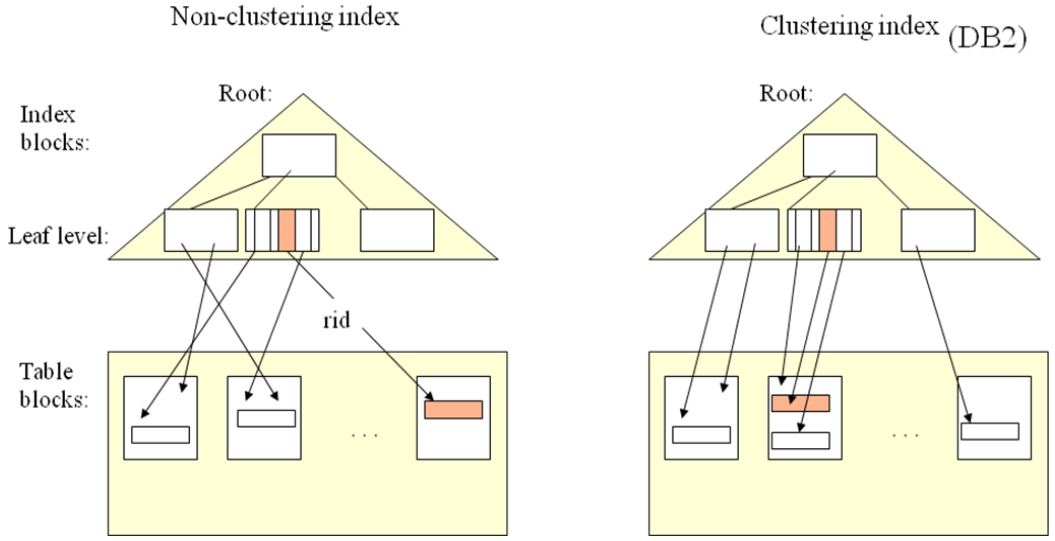
In this chapter, we look at some topics on indexes in the mainstream DBMS systems which we are interested in .

DB2 LUW

Following is a simplified syntax of CREATE INDEX. Keyword CLUSTER defines that the index is a clustering index (see figures below). The difference between clustering and clustered index is not clear for everybody, and so in some literature DB2 CLUSTER type indexes are called clustered indexes.

```
CREATE [ UNIQUE ] INDEX <index name>
ON <table> ( <column [ASC | DESC ] > [, ... ] )
[ IN <tablespace> ]
[ INCLUDE ( <column> [, ... ] ) ]
[ CLUSTER ]
[ PCTFREE <per cent> ]
[ MINPCTUSED <per cent> ]
[ [DIS]ALLOW REVERSE SCANS] ]
[ COLLECT [[ SAMPLED] DETAILED] STATISTICS ]
[ COMPRESS [{YES | NO}] ]
```

PCTFREE and COLLECT STATISTICS options affect the process when index is created to a table which already contains rows.



To study the application plans in DB2 one has to create special Explain tables [3] using the script explain.ddl in CLP command window

```
db2 -tvf explain.ddl
```

then using the explain tool

```
explain plan with snapshot for "<query>"
```

this will store the explain data into Explain tables from which the plan can be printed using the tool **db2exfmt** or the **Visual Explain** tool from Command editor.

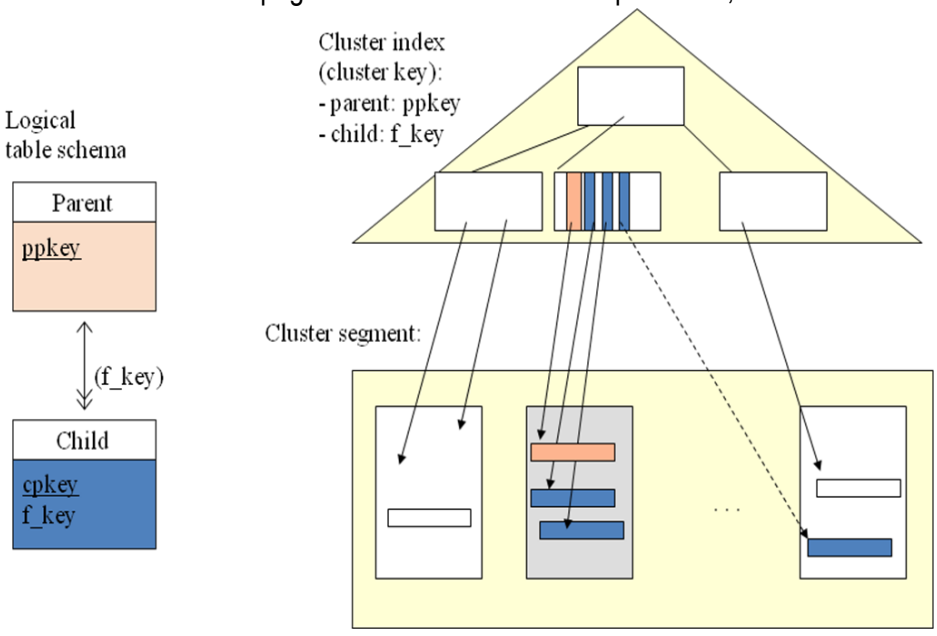
Oracle

Following is a simplified syntax of Oracle's CREATE INDEX for a table

```
CREATE [ UNIQUE | BITMAP]
INDEX [<schema>.]<index name>
ON [<schema>.]<table>
    ( <column_expr> [ASC | DESC ] > [, … ] )
[ PCTFREE <per cent> ] [ PCTUSED <per cent> ]
[ LOGGING | NOLOGGING ]
[ COMPUTE STATISTICS ]
[ TABLESPACE <tablespace> ]
[ COMPRESS <per cent> | NOCOMPRESS]
[NOSORT | REVERSE ]
```

With NOLOGGING we suppress logging of index creation and maintenance operations, and data load operations.

Beside indexes on tables, Oracle supports Index-organized tables (IOT) similar to clustered indexes of SQL Server, and also special Cluster segment for which a special cluster index is created and in which a page can contain rows of multiple tables, which share the cluster index.



SQL Server

Following is a simplified syntax of CREATE INDEX in Transact-SQL

```
CREATE [UNIQUE] [CLUSTERED | NONCLUSTERED]
INDEX <index_name> ON <table> (<column> [,...])
[INCLUDE (<column> [,...]) ]
[WITH
        [PAD_INDEX]
        [[,] FILLFACTOR = <fillfactor>]
        [[,] IGNORE_DUP_KEY]
        [[,] DROP_EXISTING]
        [[,] STATISTICS_NORECOMPUTE]
]
[ON <filegroup>]
```

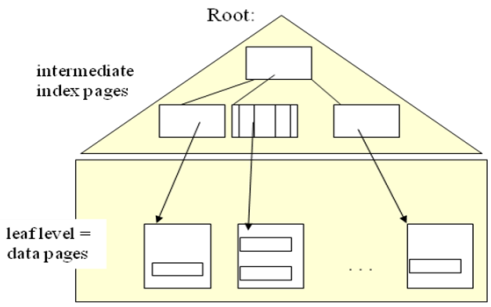
For details, we refer to SQL Server Books Online.

As default, SQL Server creates a **clustered index** for the Primary Key constraint

Clustered Index

- Leaf nodes are on table pages and the data rows on these are always in order
- Sparse index (on the lowest level of intermediate index pages)
- Depth of the index grows adding new levels just above the leaf node level

see Delaney: Inside SQL Server



Gulutzan and Pelzer call this as "Strong-Clustered Index"

In this case, on index records of all alternate indexes instead of RIDs, SQL Server uses "bookmarks" as row addresses, which are the primary key values of the rows pointing to the clustered index of the primary key. The clustered index can also be some non-unique index, in which case the address is the index record key extended by an internal 4 byte uniqueifier part, and these keys are used as bookmarks in the alternate indexes pointing to the clustered index.

Table with Unique Clustered Index

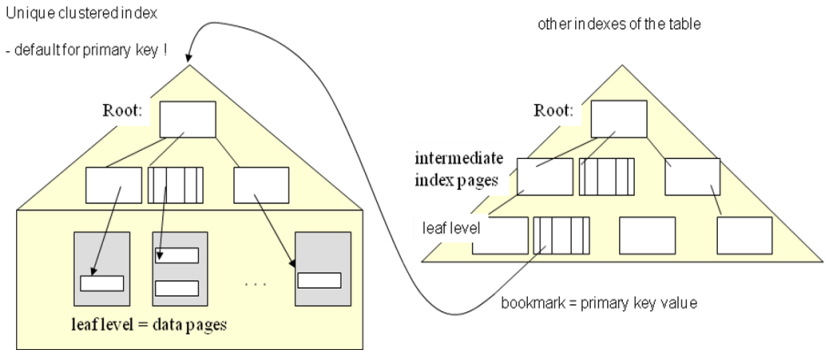


Table with non-unique Clustered Index

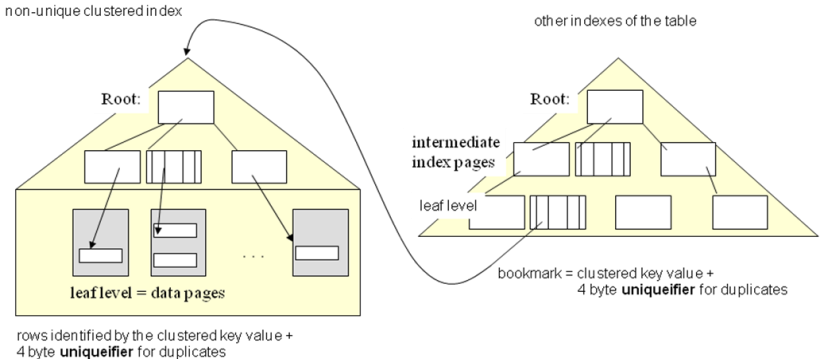
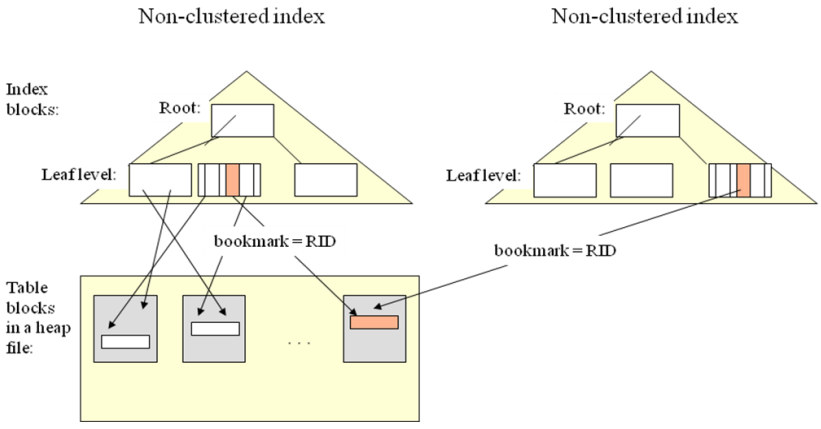


Table without clustered index



Part III - Index Design and Performance Labs

Software to be used and the Learning Environment alternatives:

a) Own environment and DBMS instances

We will focus on the mainstream DBMS systems: DB2, Microsoft SQL Server, and Oracle. You may already have some of these available, but if not, you may download and install the latest free "Express" editions of these on your computer, for example

- DB2 Express-C you may download at http://www-01.ibm.com/software/data/db2/express/ (or look for the current site)
- Microsoft SQL Server Express you will find starting at http://www.microsoft.com/sql
- Oracle XE you will find starting at http://otn.oracle.com

For all these systems, you will also find the installing and "start to use" instructions on the corresponding download sites.

b) DBTLab on Linux **(ref 19)**

To make at least part of the materials of the popular DBTech Pro live workshops available to public, to schools, individual students and database professionals in their own Life-Long Programmes, we provide downloadable private labs built of free or open source software and free professional, mainstream database systems, such as DB2 Express-C of IBM and Oracle XE of Oracle, available on multiple operating system platforms, including free Linux platforms. SQL Server Express of Microsoft is also freely available, but since it is available only on the proprietary Windows platforms of Microsoft, we cover its features and examples in our tutorials, but users need to build the SQL Server labs of their own.

These are the current downloadable Database Labs:

- `www.DBTechNet.org/download/VMware_SUSE_DB2Lab.zip`
  Based on the free SUSE Linux Enterprise Server SLES 10 in VMware virtual machine image with DB2 Express-C available from IBM's DB2 web site, in which we have replaced DB2 Express-C V9.7 and installed the DB2INDEX database of the Index Design and Performance Labs as user `db2inst2/db2inst2`
  Note:
  The latest version of DB2 Express-C is available at IBM's web site
  `http://www-01.ibm.com/software/data/db2/express/download.html`

- `www.DBTechNet/download/Ubuntu_OracleXE.zip`
  Oracle XE 10g on Ubuntu 8.4 Linux in Microsoft Virtual PC 2007 image. The main user in this lab is `dbtech` and the initial password of the user is `dbtech`.
  Microsoft Virtual PC 2007 is freely available at Microsoft's download center
  Note: Index Design Lab VLW has not yet been ported to this lab environment.

Both DBTLabs can be used as private labs on experimenting with the appropriate examples and exercises of the following tutorials. We have tested them on Windows XP/Vista platforms, with workstations of 2GB or more RAM.

IDPLab1: Experimenting with SQL Server Indexes

SQL Server Index Lab by Kari Silpiö

For this lab we assume that you have some SQL Server edition on your workstation or some virtual computer image of your own, and that you have local administrator privileges. For example, SQL Server Express is a free edition which you can download (with some Express tools to be included in the installation) from Microsoft's web sites. SQL Server is widely used in the industry. Due to its easy but advanced tools, it is most suitable as the first real DBMS system to study.

Part 1. SQL Server Index Basics

The lab contains experimental tasks on the following:

A. Storage Allocation
B. Indexes and Performance
C. Index Properties and Fragmentation

Outcomes:
You will have hands-on experiences on how indexes improve performance, index properties and index fragmentation. Learn how to view physical properties of indexes and minimize fragmentation by rebuilding indexes.

Start with the document
http://www.DBTechNet.org/labs/idp_lab/IDPLab1a-IndexBasics.pdf
You can create the test database for the lab with the following script file
http://www.DBTechNet.org/labs/idp_lab/IDPLab1-CreateDatabase.sql
The additional handout 'Quick Reference for the SQL Server Index Lab' provides you more information on the commands you have to execute in the lab.
http://www.DBTechNet.org/labs/idp_lab/IDPLab1-QuickReference.pdf

Part 2. SQL Server Database Engine Tuning Advisor

In this lab you experiment with the Database Tuning Advisor for choosing indexes.

Outcomes:
You will learn the basics on using an index advisor for choosing indexes and have hands-on experiences on using SQL Server Database Tuning Advisor. You will know how to use SQL Server Profiler to create traces that you can use as workloads to be analyzed with the Tuning Advisor.

If you have a full SQL Server edition[1] (Enterprise, Standard, Developer, or Evaluation[2]), then you can proceed with the tasks at http://www.dbtechnet.org/labs/idp_lab/IDPLab1b-TuningAdvisor.pdf

[1] SQL Server Database Tuning Advisor is not included in the free SQL Server Express edition.
[2] You can download the SQL Server Enterprise 180-day evaluation version from Microsoft's pages.

IDPLab2: Verifying QUBE Estimates

In the spring at 2004, in the previous project DBTech Pro of DBTechNet. we had a most exciting subproject preparing the Index Design workshop presented by Tapio Lahdenmäki at Malaga University, Spain. Tapio Lahdenmäki designed a simplified database of 2 tables, Customers and Invoices, with given distributions of column values, and a set of query scenarios (Step 1A,…) with planned indexes and calculated QUBE estimates to be tested in hands-on-workshop. According to the plan, the exercises should not take too long, but cardinalities of the tables should be big enough so that students could experience the effects of proper indexing on elapsed times. So we figured that 1 million might be suitable number of customers and 4 invoices per customer, which makes 4 million invoices.

We generated the test contents on a SQL Server 2000 instance. Then the table contents were ported to various DBMS platforms creating about the same test sets on Oracle 9 instances, DB2 LUW 8 and DB2 for z/OS at some companies and schools round the southern Finland. Based on our experiences, the lab was set up by Jaakko Rantanen using the multiple remote Oracle server instances at Hamk, Häme Polytechnic in Hämeenlinna. The problem in this arrangement was that the students were not able to create indexes of their own, but they could only switch between the fixed index configurations in those ready Oracle instances.

The current virtual computer technologies make it now possible to arrange the workshop material available for private labs where the users can experiment also with data loadings, creating and dropping indexes themselves according to the personal decisions.

Scenarios to be tested

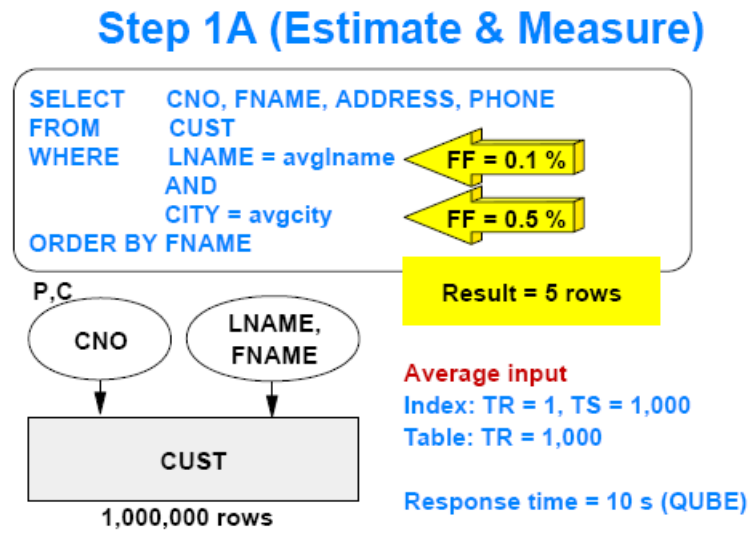
By permission of Tapio Lahdenmäki we have copied the following figures defining the query scenarios (Step 1A .. Step 6C) from the original Index Design Lab plan for Helia (1.2.2004). The figures present us the setup of indexes and the pessimistic elapsed time estimate of QUBE method for the queries to be verified by measurements on running the queries.

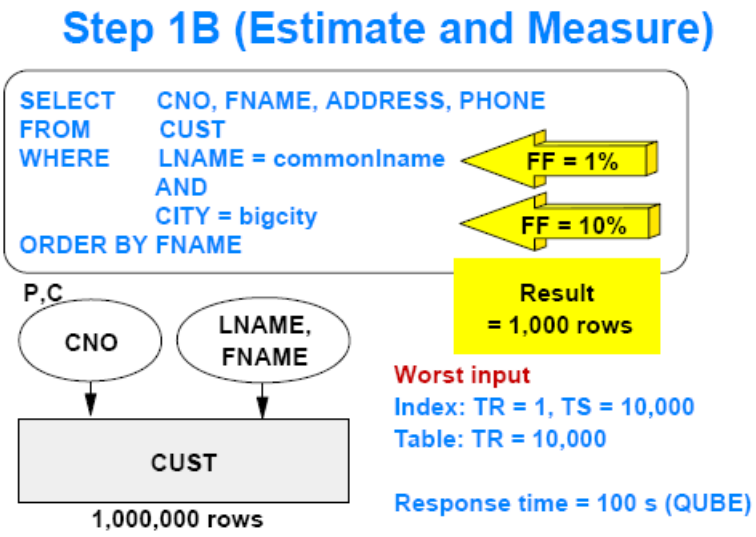
The legend in the figures

- table presented as a box, and circles present indexes
- P = index of the primary key
- C = clustering index
- FF = filter factor of the predicate

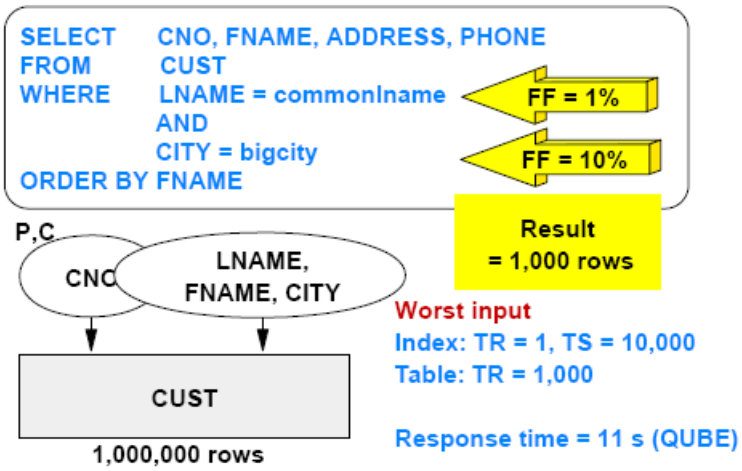
Since the scenarios assume clustering indexes on customer numbers (CNo), the test data for the tables is sorted on CNo columns (..Rand.dat files). So the clustering indexes will be in ideal condition with no fragmentation at all.

Following picture presents scenario for Step 1A - example of inadequate indexing for the query

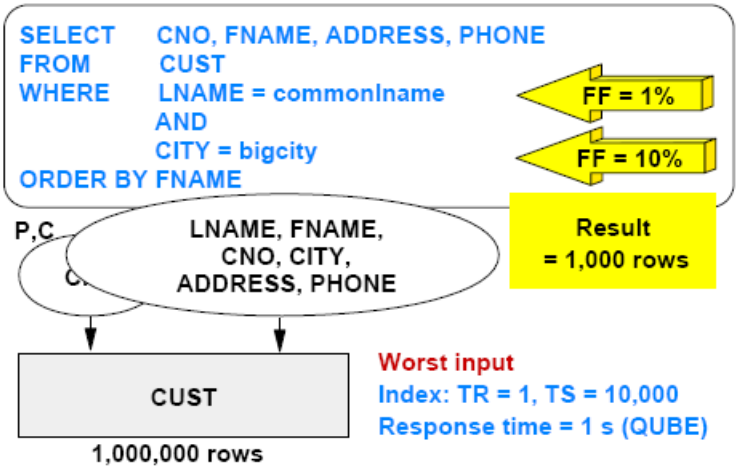




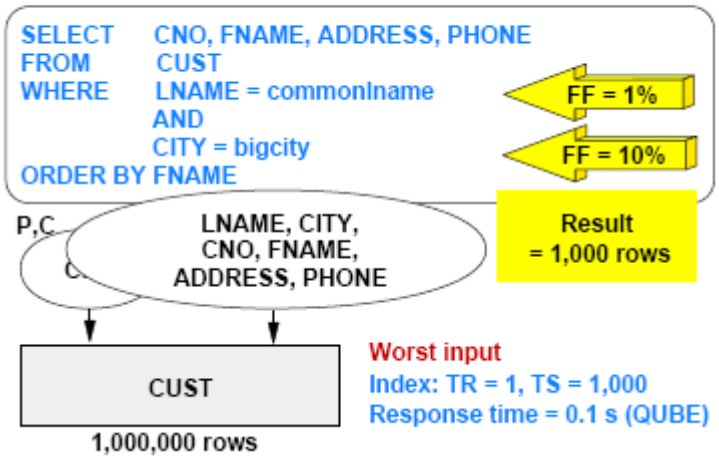
Step 2A (Estimate and Measure)



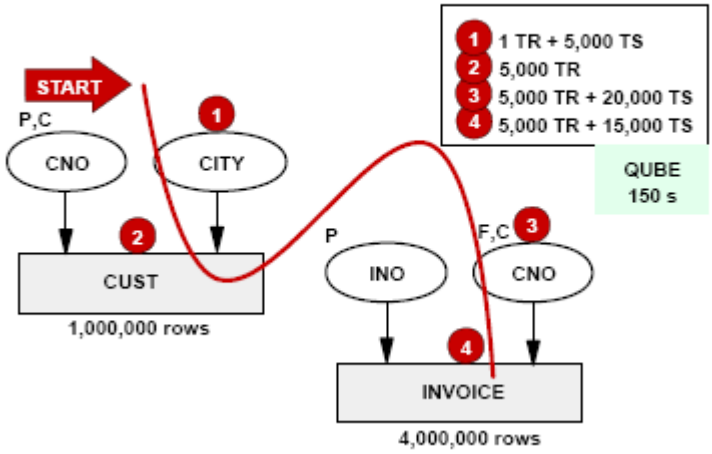
Step 2B (Estimate and Measure)



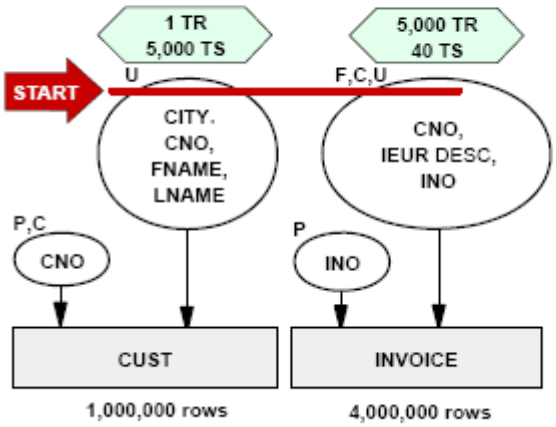
Step 2C (Estimate and Measure)



Step 3 - Average City (5,000 Cust)



Step 4A - Average City (5,000 Cust)



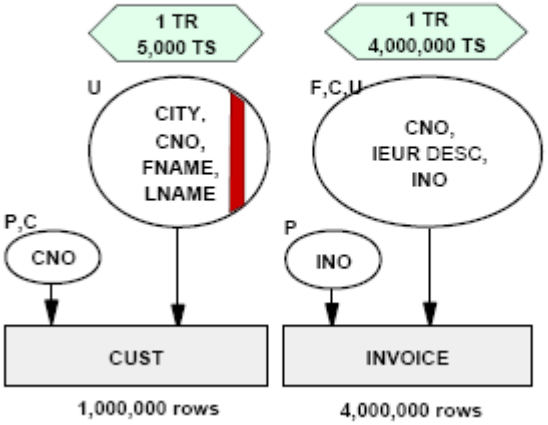
QUBE
50 s

40 big invoices (0.2 % of 20,000) - thus, 40 TS

(MC=2)

Step 4A - Average City (5,000 Cust)

Merge scan or hash join

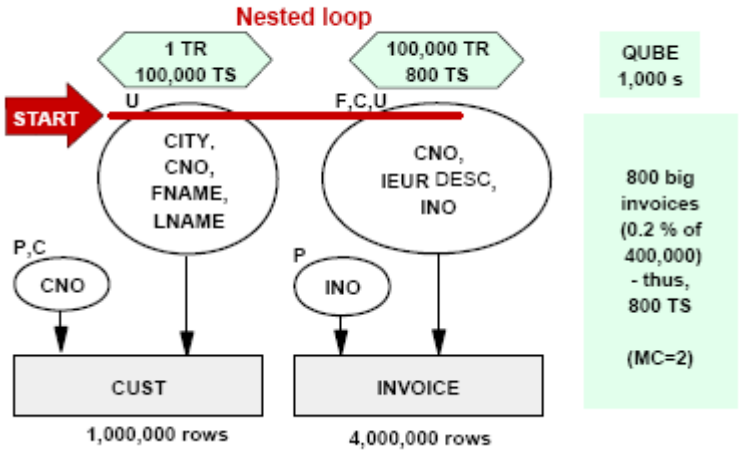


QUBE
400 s

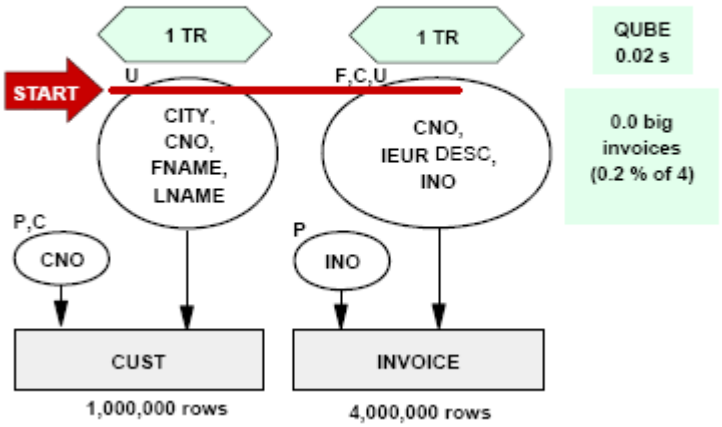
But full index scan much faster as almost all rows rejected: 3,998,000 TL a 0.01 ms = 40 s

(TL = light touch)

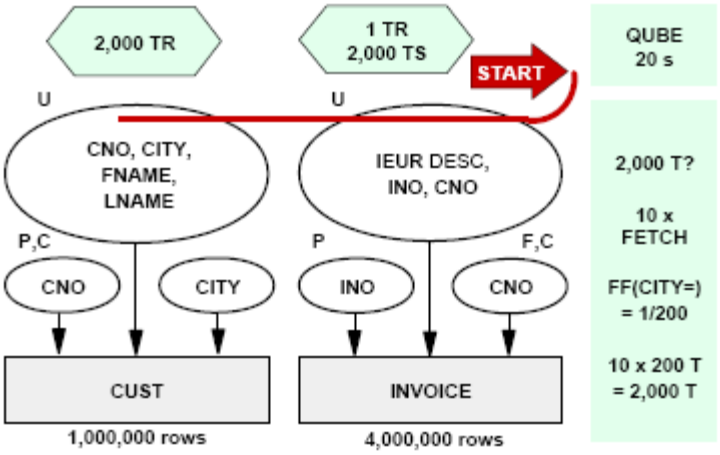
Step 4B - Big City (100,000 Cust)



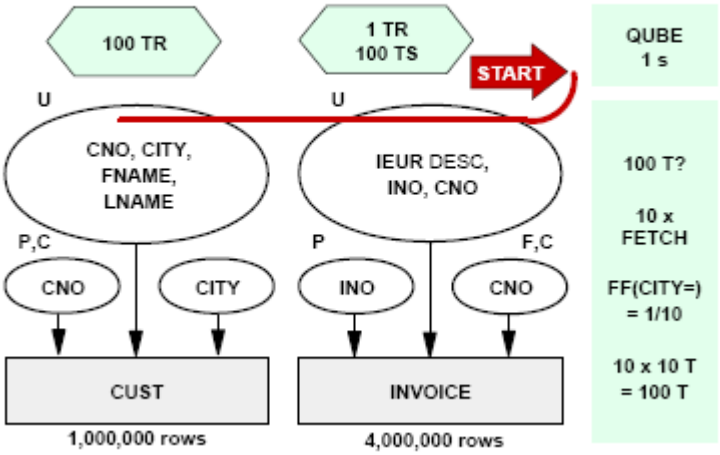
Step 4C - Small City (1 Cust)



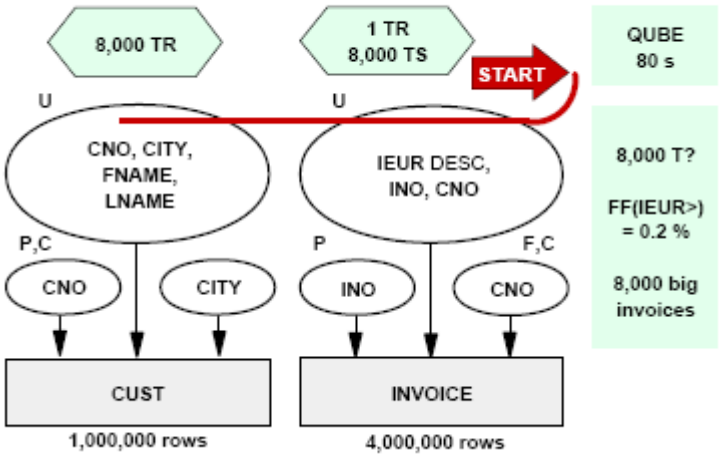
Step 5A - Average City (5,000 Cust)



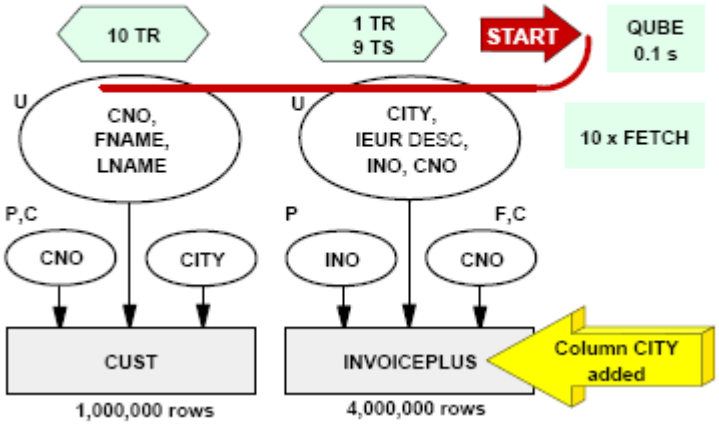
Step 5B - Big City (100,000 Cust)



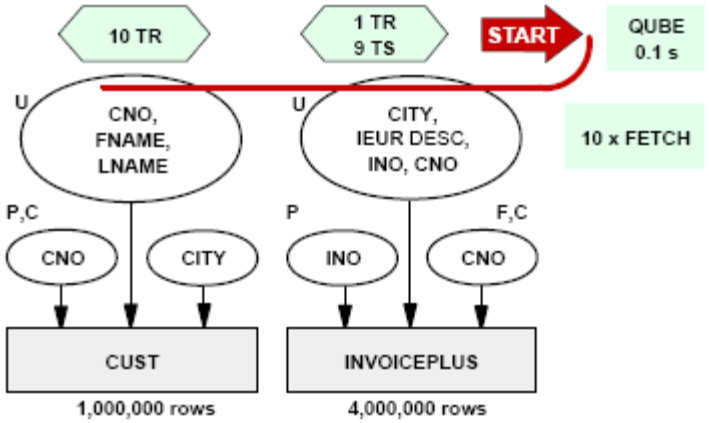
Step 5C - Small City (1 Cust)



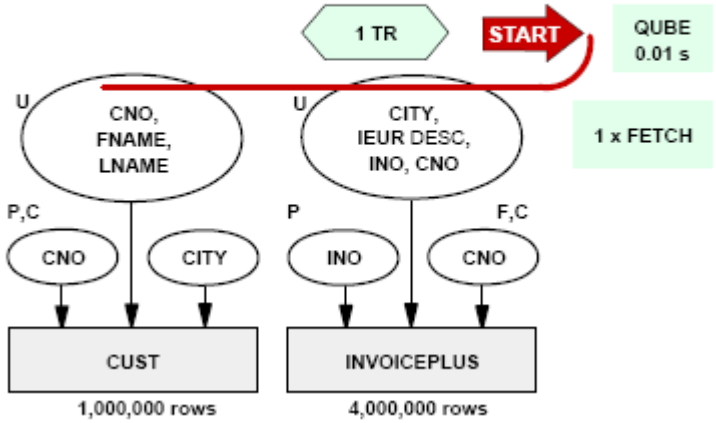
Step 6A - Average City (5,000 Cust)



Step 6B - Big City (100,000 Cust)



Step 6C - Small City (1 Cust)



and in the following the Step scenarios are presented as SQL commands of DB2 LUW preceded by preparation of the indexes for the step.

```
-- Index Design Lab
-- Steps for verifying corresponding CUBE estimates
--
-- Step 1A: 5 Rows, QUBE 10s

CREATE INDEX Cust_X1 ON Cust(lname, fname)
PCTFREE 10 ALLOW REVERSE SCANS
COLLECT STATISTICS ;

      SELECT cno, fname, address, phone
      FROM Cust
      WHERE lname = 'Lname287' -- FF = 0.1 %
        AND city = 'Truro'     -- FF = 0.5 %
      ORDER BY fname;

-- Step 1B: 1000 Rows, QUBE 100s

      SELECT cno, fname, address, phone
      FROM Cust
      WHERE lname = 'Adams'
      AND city = 'BigCity'
      ORDER BY fname;

-- Step2A: 1000 Rows. QUBE 11s

DROP INDEX Cust_X1 ;
CREATE INDEX Cust_X2 ON Cust(lname,fname,city)
PCTFREE 10 ALLOW REVERSE SCANS
COLLECT STATISTICS ;

      SELECT cno, fname, address, phone
      FROM Cust
      WHERE lname = 'Adams'
      AND city = 'BigCity'
      ORDER BY fname;

-- Step 2B  1000 Rows. QUBE 1s

DROP INDEX Cust_X2 ;
CREATE INDEX Cust_X3 ON Cust(lname,fname,cno,city,address,phone)
PCTFREE 10 ALLOW REVERSE SCANS
COLLECT STATISTICS ;

      SELECT cno, fname, address, phone
      FROM Cust
      WHERE lname = 'Adams'
      AND city = 'BigCity'
      ORDER BY fname;

-- Step 2C  1000 Rows. QUBE 0,1s

DROP INDEX Cust_X3 ;
```

```
CREATE INDEX Cust_X4 ON Cust(lname,city,cno,fname,address,phone)
PCTFREE 10 ALLOW REVERSE SCANS
COLLECT STATISTICS ;

      SELECT cno, fname, address, phone
      FROM Cust
      WHERE lname = 'Adams'
      AND city = 'BigCity'
      ORDER BY fname;

-- Step 3   Average city (5000 cust) QUBE 150s

RUNSTATS ON TABLE DB2INST2.INVOICE ON KEY COLUMNS AND
INDEX DB2INST2.INVOICE_PK ALLOW WRITE ACCESS ;
COMMIT ;

DROP INDEX Cust_X4 ;
CREATE INDEX Cust_X5 ON Cust(city)
PCTFREE 10 ALLOW REVERSE SCANS
COLLECT STATISTICS ;

CREATE INDEX Invoice_FK ON Invoice(cno) CLUSTER
PCTFREE 10 ALLOW REVERSE SCANS
COLLECT STATISTICS ;

      SELECT lname, fname, cust.cno, ino, ieur
      FROM cust INNER JOIN invoice ON
       cust.cno = invoice.cno
      WHERe cust.city = 'Truro'
      AND ieur > 50000
      ORDER BY ieur desc;

-- Step 4A  Average city (5000 cust) QUBE 50...400 s

DROP INDEX Cust_X5 ;
CREATE INDEX Cust_X6 ON Cust(city,cno,fname,lname)
PCTFREE 10 ALLOW REVERSE SCANS
COLLECT STATISTICS ;

DROP INDEX Invoice_FK;
CREATE INDEX Invoice_FK2 ON Invoice(cno,ieur DESC,ino) CLUSTER
PCTFREE 10 ALLOW REVERSE SCANS
COLLECT STATISTICS ;

      SELECT lname, fname, cust.cno, ino, ieur
      FROM cust INNER JOIN invoice
      ON cust.cno = invoice.cno
      WHERE cust.city = 'Sunderland'
      AND ieur > 50000
      ORDER BY ieur desc;

-- Step 4B  big city (100000 cust) QUBE 1000s

      SELECT lname, fname, cust.cno, ino, ieur
      FROM cust INNER JOIN invoice
      ON cust.cno = invoice.cno
      WHERE cust.city = 'BigCity'
```

```
      AND ieur > 50000
      ORDER BY ieur desc;

-- Step 4C  small city (1 cust) QUBE 0,02s

      SELECT lname, fname, cust.cno, ino, ieur
      FROM cust INNER JOIN invoice
      ON cust.cno = invoice.cno
      WHERE cust.city = 'SmallCity'
      AND ieur > 50000
      ORDER BY ieur desc;

-- Step 5A: Average city (5000 cust) QUBE 20s

DROP INDEX Cust_X6 ;
CREATE INDEX Cust_U1 ON Cust(cno,city,fname,lname)
PCTFREE 10 ALLOW REVERSE SCANS
COLLECT STATISTICS ;

CREATE INDEX Cust_City ON Cust(city)
PCTFREE 10 ALLOW REVERSE SCANS
COLLECT STATISTICS ;

DROP INDEX Invoice_FK2;
CREATE UNIQUE INDEX Invoice_U1 ON Invoice(ieur DESC,cno,ino)
PCTFREE 10 ALLOW REVERSE SCANS
COLLECT STATISTICS ;

CREATE INDEX Invoice_FK3 ON Invoice(cno) CLUSTER
PCTFREE 10 ALLOW REVERSE SCANS
COLLECT STATISTICS ;

      SELECT lname, fname, cust.cno, ino, ieur
      FROM cust INNER JOIN invoice ON
      cust.cno = invoice.cno
      WHERE cust.city = 'Sunderland'
      AND ieur > 50000
      ORDER by ieur desc;

-- Step 5B  Big city (100000 cust) QUBE 1s

      SELECT lname, fname, cust.cno, ino, ieur
      FROM cust INNER JOIN invoice ON
      cust.cno = invoice.cno
      WHERE cust.city = 'BigCity'
      AND ieur > 50000
      ORDER by ieur desc;

-- Step 5C  Small city (1 cust) QUBE 80s

      SELECT lname, fname, cust.cno, ino, ieur
      FROM cust INNER JOIN invoice ON
      cust.cno = invoice.cno
      WHERE cust.city = 'SmallCity'
      AND ieur > 50000
      ORDER by ieur desc;
```

```
-- Step 6A: Average city (5000 cust) QUBE 0,1s

ALTER TABLE InvoicePlus ADD
CONSTRAINT InvoicePlus_FK
FOREIGN KEY (CNo) REFERENCES Cust (Cno) ;

CREATE INDEX InvoicePlus_FK
ON INVOICEPLUS (CNO ASC) CLUSTER
PCTFREE 10 ALLOW REVERSE SCANS
PAGE SPLIT SYMMETRIC
COLLECT STATISTICS ;

RUNSTATS ON TABLE DB2INST2.INVOICEPLUS ON KEY COLUMNS AND
INDEX DB2INST2.INVOICE_PK ALLOW WRITE ACCESS ;
COMMIT ;

DROP INDEX Cust_U1 ;
CREATE INDEX Cust_U2 ON Cust(cno,fname,lname)
PCTFREE 10 ALLOW REVERSE SCANS
COLLECT STATISTICS ;

CREATE UNIQUE INDEX InvoicePlus_U1
    ON InvoicePlus(city,ieur DESC,ino,cno)
PCTFREE 10 ALLOW REVERSE SCANS
COLLECT STATISTICS ;

CREATE INDEX InvoicePlus_FK ON InvoicePlus(cno) CLUSTER
PCTFREE 10 ALLOW REVERSE SCANS
COLLECT STATISTICS ;

      SELECT lname, fname, cust.cno, ino, ieur
      FROM cust INNER JOIN invoiceplus ON
      cust.cno = invoiceplus.cno
      WHERE cust.city = 'Truro'
      AND ieur > 50000
      ORDER by ieur desc;

-- Step 6B  Big city (100000 cust) QUBE 0,1s

      SELECT lname, fname, cust.cno, ino, ieur
      FROM cust INNER JOIN invoiceplus ON
      cust.cno = invoiceplus.cno
      WHERE cust.city = 'BigCity'
      AND ieur > 50000
      ORDER by ieur desc;


-- Step 6C  small city (1 cust) QUBE 0,01s

      SELECT lname, fname, cust.cno, ino, ieur
      FROM cust INNER JOIN invoiceplus ON
      cust.cno = invoiceplus.cno
      WHERE cust.city = 'SmallCity'
      AND ieur > 50000
      ORDER by ieur desc;

-- Step7A   Average city (5000 cust)
```

```
DROP INDEX Cust_U2 ;
DROP INDEX Cust_CITY ;
DROP INDEX Cust_U3 ;
CREATE UNIQUE INDEX Cust_U3 ON Cust(city,cno,lname,fname)
PCTFREE 10 ALLOW REVERSE SCANS
COLLECT STATISTICS ;
--
DROP INDEX Invoice_U1;
CREATE UNIQUE INDEX Invoice_U2 ON Invoice(ieur DESC,ino,cno)
PCTFREE 10 ALLOW REVERSE SCANS
COLLECT STATISTICS ;

      SELECT lname, fname, cust.cno, ino, ieur
      FROM cust INNER JOIN invoice ON
      cust.cno = invoice.cno
      WHERE cust.city = 'Sunderland'
      AND ieur > 50000
      ORDER BY ieur desc

--Step 7B   Big city (100000 cust)

      SELECT lname, fname, cust.cno, ino, ieur
      FROM cust INNER JOIN invoice ON
      cust.cno = invoice.cno
      WHERE cust.city = 'BigCity'
      AND ieur > 50000
      ORDER BY ieur desc

-- Step 7C  small city (1 cust)

      SELECT lname, fname, cust.cno, ino, ieur
      FROM cust INNER JOIN invoice ON
      cust.cno = invoice.cno
      WHERE cust.city = 'SmallCity'
      AND ieur > 50000
      ORDER BY ieur desc
```

See Appendix 1 for advice on running this lab on the VMware_SUSE_DB2Lab virtual computer available on our Web site.

IDPLab3: Create Index exercises using DB2 Express-C

Warning: You need some experience of the IDPLab2 before operating with DB2 and Linux platform in these lab works!

See the history of the database DB2INDEX in IDPLab2.
We suggest that you make a copy of the VMware-SUSE computer for this lab and make use of the following files

In the folder db2indexfiles you will find following files
- Invoices_generic.txt - historic copy of creation of the test data first on SQL Server
- CustRand.dat - 1 M customers in random order
- InvoicePlusRand.dat - 4 M invoicePlus rows in random order
- CustSort.dat - 1 M customers sorted in Cno order (used in IDPLab2)
- InvoicePlusSort.dat - 4 M invoicePlus sorted in Cno order (used in IDPLab2)

and for testing the 50 customer sample files with header lines:
- Cust50Rand.dat
- InvoicePlus50Rand.dat

In the home directory of db2inst2 you will find the script file `import` from which you can modify scripts for data importing exercises in this lab3.

For the tasks of this lab we just give you the tasks, and you should figure out yourself how to work these out in the Linux platform.
Use **`db2batch`** tool to measure the requested times. Use db2batch –h for documentation of the tool.

Task a) Measuring the time of loading the data without indexes
- Drop the indexes and recreate the tables without primary keys and foreign keys
- Measure the time of loading the data
- Measure the time of creating primary keys
- Measure the time of creating the foreign key (cno) of invoice to cust (cno)
- Measure the time of creating index for the foreign key

Task b) Measure the time of loading the data with indexes
- Drop the indexes and recreate the tables with primary keys and foreign keys
- Create index for the foreign key before loading the data
- Measure the time of loading the data

Compare the net elapsed times of tasks a and b.
What kind of risks do we have in task a?

Task c) Tradeoff FF point on index scan and table scan?

- This is a difficult, but an interesting question!

The question is: What is the lowest Filter Factor % when using a table scan on CUST table is faster than using index scan on the leaf level of the following index

```
CREATE INDEX Cust_City ON Cust (CITY)
```

for the query

```
SELECT COUNT(*)
FROM  Cust
WHERE city = :city
```

The first problem is to find distribution of the cities to find the city names which you want to experiment the test runs

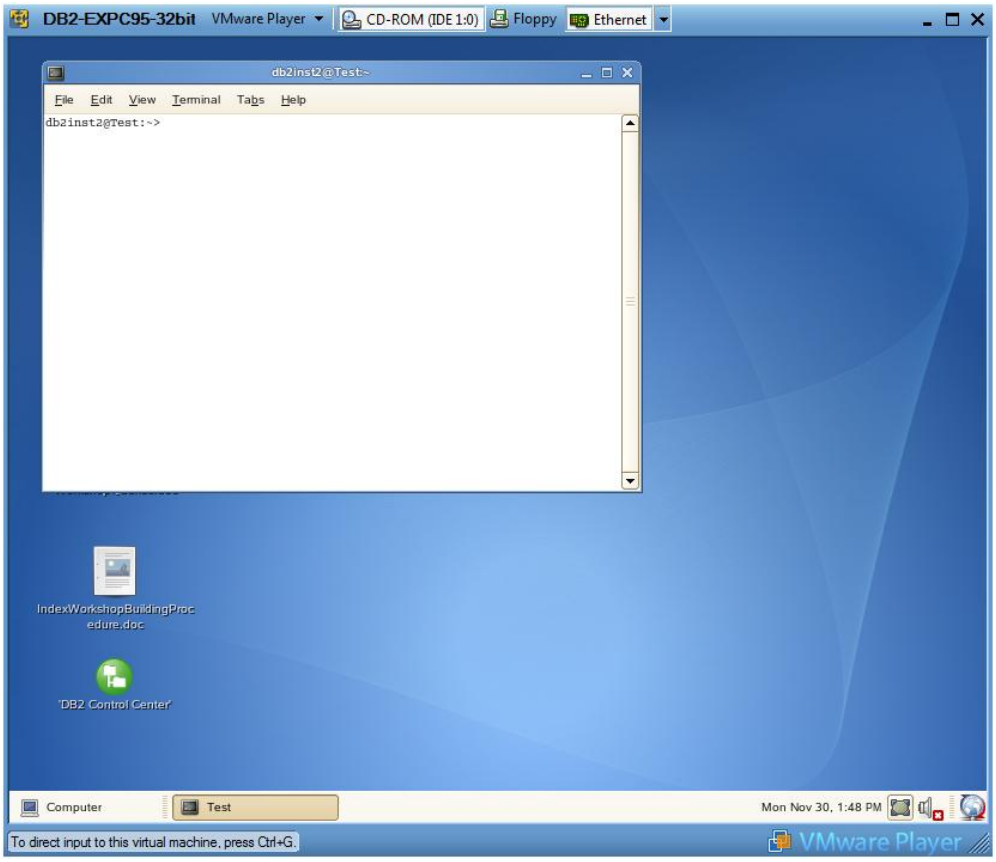
```
SELECT city, count(*) as customers
FROM Cust
GROUP BY city
HAVING count(*) BETWEEN 500 and 3000
```

.
and then starting from the city which is near 3% (i.e. about 3000) of customers proceeding with "binary search" procedure to find the proper FF.
Does the optimizer find the proper plan or do we need to force the optimizer?
The elapsed time measured by db2batch is the proper criteria for the tradeoff point.

Appendix 1 IDBLab2 scripts for DB2 Express-C

We have modified parts of this lab to a self-study virtual lab, first implemented on the free SUSE on VMware virtual computer provided by IBM in which environment we at Haaga-Helia as student project have replaced DB2 Express-C 9.5 with version 9.7, and installed the tables in DB2INDEX database for user db2inst2 (the same as password).

Be patient with this virtual computer. Even on fairly powerful workstations, it takes time to load. After some 3-4 minutes you will see the login screen. Enter 'db2inst2' as the user id and same as the password. Again wait some1-3 minutes to see the graphical interface of GNOME, and terminal window at home directory of user db2inst2



From Computer - "More tools" of GNOME you will find a rich set of tools, including DB2 and Open Office, so you don't need to be a Linux guru to work in the lab. However, basic knowledge on Unix and Linux systems will help a lot. Introduction to those is out of scope of this paper.

We have organized the lab files in following folders of db2inst2 (under **`/home/db2inst2`**)

`./Documents` is the default folder of gedit text editor files

`./db2indexfiles` contains the original data files (*rand in random order and *cno in CNo order) to be loaded to the tables

./Sripts	here we have stored the scripts for the steps of lab scenarios, and for index maintenance
./Reports	here the scripts will write the reports of the step scenarios, and also XLS report for collecting your report of elapsed times in various steps.

The database DB2INDEX is ready for your test runs, but if you drop the database and create it again, you need to create the tables CUST, INVOICE, and INVOICEPLUS using the script
./createtables
and import contents of the tables using scripts
./import cust
./import invoice
./import invoiceplus
and setup indexes for steps 1A-1B using script
./indexes 1

As you start the testing for the first time the contents of the tables have already been loaded from the custcno.dat and invoicepluscno.dat files where the data is sorted in customer number CNO order. So all you need to do is to run the following scripts of the steps in the following order.

Please note that setting up the indexes for any steps depend always on the status of indexes in the previous step. If you try something in different order, you need to setup the indexes by yourself studying the file Scripts/Steps.

```
./runstep 1A
./runstep 1B
./indexes 2A
./runstep 2A
./indexes 2B
./runstep 2B
./indexes 2C
./runstep 2C
./indexes 3
./runstep 3
./indexes 4
./runstep 4A
./runstep 4B
./runstep 4C
./indexes 5
./runstep 5A
./runstep 5B
./runstep 5C
./indexes 6
./runstep 6A
./runstep 6B
./runstep 6C
./indexes 7
./runstep 7A
./runstep 7B
./runstep 7C
```

and take notes of the elapsed times ('Execute Time') of runstep reports, and compare how these match with the corresponding CUBE estimates.

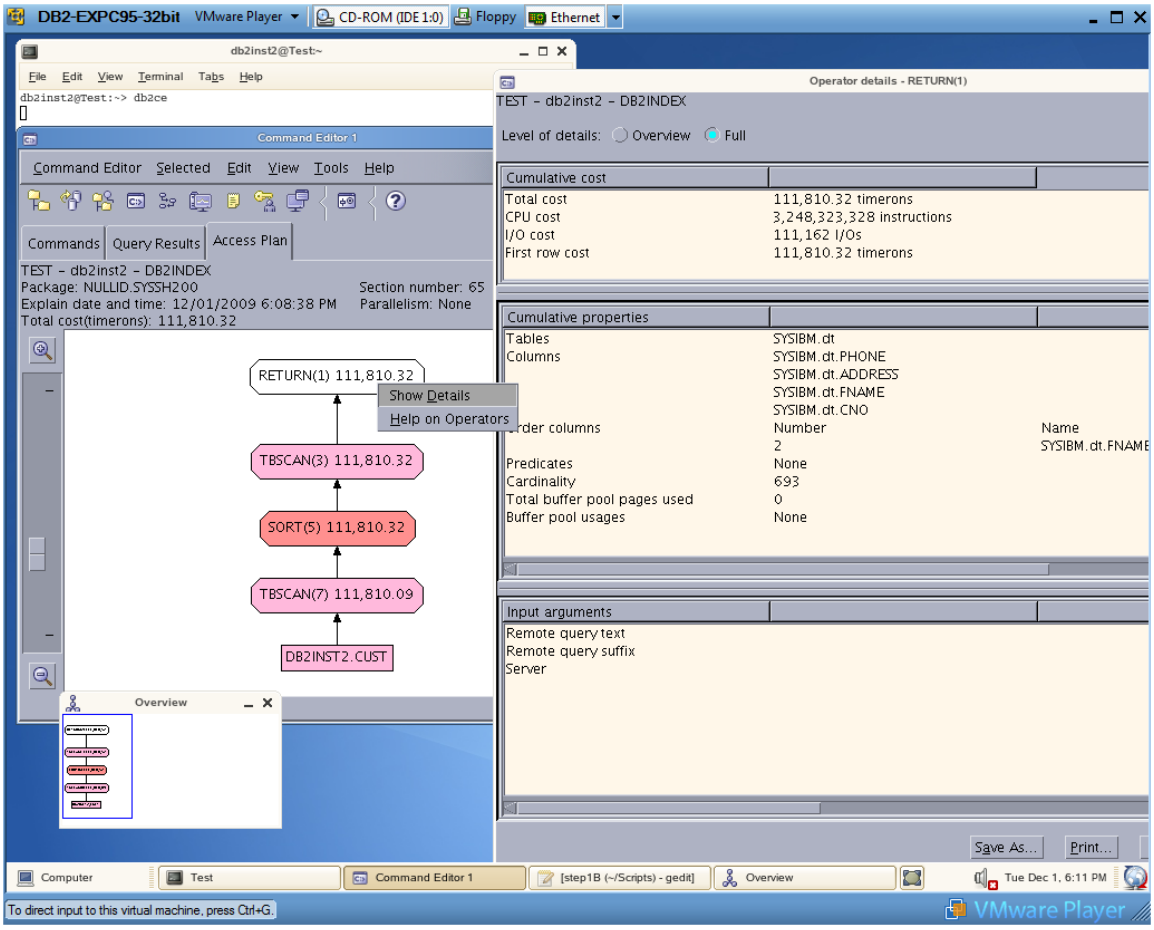
Note. Virtual computer is ideal for the Lab in terms that in the standalone environment you can do anything, and if you mess with it, you can simply delete the replace the image files with original image files, and start from the beginning. Unfortunately the performance measurements on file I/O are not totally realistic: - since the **discs of the virtual computer are files of the host system**, we cannot bypass the file system cache of the host, and after first measurements, even if we stop our DB2 instance, the repeated test runs will be faster than the first run, as you can see from the following test runs:

```
db2inst2@Test:~> ./runstep 7A
Index Design Lab Step7A
Mon Nov 30 14:13:30 EET 2009
- Stopping DB2 instance to clear the bufferpool
DB20000I  The TERMINATE command completed successfully.
SQL1064N  DB2STOP processing was successful.
SQL1063N  DB2START processing was successful.
- Run step7A reporting elapsed times
DB2 Universal Database Version 9.7, 5622-044 (c) Copyright IBM Corp. 1991, 2008
Licensed Material - Program Property of IBM
IBM DATABASE 2 Explain Table Format Tool

Connecting to the Database.
Connect to Database Successful.
Output is in Reports/step7A.out.
Executing Connect Reset -- Connect Reset was Successful.
DB20000I  The TERMINATE command completed successfully.
Please mark the execution time to report at Reports/IDLSteps.xls
* Prepare Time is:       2.732011 seconds
* Execute Time is:       0.487793 seconds
* Fetch Time is:         0.000210 seconds
* Elapsed Time is:       3.220014 seconds (complete)
db2inst2@Test:~> ./runstep 7A
Index Design Lab Step7A
Mon Nov 30 14:16:49 EET 2009
- Stopping DB2 instance to clear the bufferpool
DB20000I  The TERMINATE command completed successfully.
SQL1064N  DB2STOP processing was successful.
SQL1063N  DB2START processing was successful.
- Run step7A reporting elapsed times
DB2 Universal Database Version 9.7, 5622-044 (c) Copyright IBM Corp. 1991, 2008
Licensed Material - Program Property of IBM
IBM DATABASE 2 Explain Table Format Tool

Connecting to the Database.
Connect to Database Successful.
Output is in Reports/step7A.out.
Executing Connect Reset -- Connect Reset was Successful.
DB20000I  The TERMINATE command completed successfully.
Please mark the execution time to report at Reports/IDLSteps.xls
* Prepare Time is:       1.581291 seconds
* Execute Time is:       0.388518 seconds
* Fetch Time is:         0.000201 seconds
* Elapsed Time is:       1.970010 seconds (complete)
db2inst2@Test:~>
```

So our test results are comparable only with other steps in this environment.

Execution plans

The following sample of Step1B (BigCity) shows an example of the Visual Plan tool of DB2 LUW can be used to display graphically the Explain Plan contents



and the following sample from Reports/step1B.rep presents the character mode presentation of Explain Plan tables from our Step1B run

step1B.rep

```
                                )
ORDER BY Q1.FNAME

Access Plan:
-----------
        Total Cost:             33583.9
        Query Degree:           1
        Cumulative Total Cost:          33583.9
        Cumulative CPU Cost:            7.94973e+07
        Cumulative I/O Cost:            4478.45
        Cumulative Re-Total Cost:       6.10606
        Cumulative Re-CPU Cost:         2.15452e+07
        Cumulative Re-I/O Cost:         0
        Cumulative First Row Cost:      73.8933
        Estimated Bufferpool Buffers:   4479.45

            Rows
           RETURN
           (   1)
            Cost
             I/O
             |
             693
           FETCH
           (   2)
           33583.9
           4478.45
          /---+----\
       7000         1e+06
      IXSCAN   TABLE: DB2INST2
      (   3)        CUST
      111.744        Q1
      53.422
        |
       1e+06
  INDEX: DB2INST2
      CUST_X1
        Q1
```

Unit of the costs above is DB2's internal cost unit TIMERON with special weights for CPU time and I/O operations.

DB2 Index Advisor tool for Workloads

If the performance of some queries is not satisfactory we may try to use Index Tuning Advisors which come with DB2. From Control Center, we can start the Design Advisor tool, but still in version 9.7 it does not seem to work properly. However, the character mode tool db2advis works fine. You will get the necessary documentation with the command

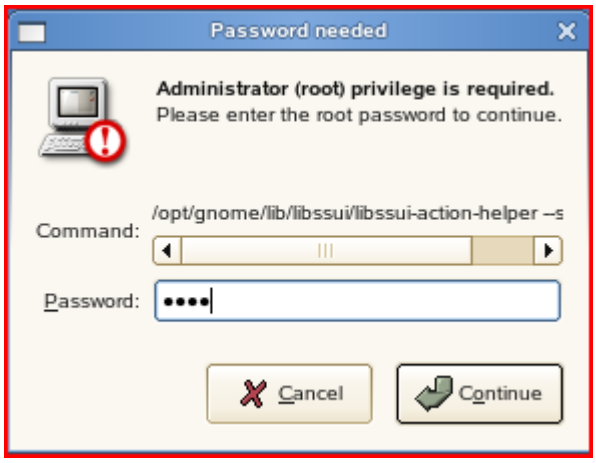
```
db2advis -h
```

and, for example, with command

```
db2advis -d DB2INDEX -i Scripts/step6A -t 5 -n db2inst2
```

you will get new recommendation of the indexes for Step6A.

Note: At shutdown of SUSE, you will be prompted to give the root user's password



and the password is 'root'.

Appendix 2 IDBLab2 Sample run of Step1A using Oracle 9.2

```
Just for comparison

-- Index Design Lab using Oracle 9.2
-- 13.10.2006 ML
--
-- Connect as SYS user and create the PLUSTRACE role as follows
DROP ROLE PLUSTRACE;
CREATE ROLE PLUSTRACE;
GRANT SELECT ON V_$SESSTAT TO PLUSTRACE;
GRANT SELECT ON V_$STATNAME TO PLUSTRACE;
GRANT SELECT ON V_$MYSTAT TO PLUSTRACE;

GRANT PLUSTRACE TO DBA WITH ADMIN OPTION;
```

```
-- Connect as SYSTEM user
-- and create a new tablespace
CREATE TABLESPACE DBTechTS
DATAFILE 'C:\Oracle\product\10.2.0\oradata\orcl\DBTECH.TBF' SIZE 1100M
AUTOEXTEND ON NEXT 50M MAXSIZE UNLIMITED;
--
--
CREATE USER DBTech
ALTER USER DBTech QUOTA 10000 M ON DBTECHTS;
GRANT CONNECT, RESOURCE, PLUSTRACE TO DBTech;

-- Connect as DBTech user
-- and create tables Cust and InvoicePlus
timing start
DROP TABLE Cust;
CREATE TABLE Cust ( -- Customers
CNo    CHAR (8) NOT NULL ,
LName  CHAR (15) , -- LastName, cardinality 1000
FName  CHAR (15) , -- FirstName, cardinality 1000
Sex    CHAR (1) , -- Contact sex
-- M=mafe,F=female,N=N/A
City   CHAR (20) , -- cardinality 200
CType  CHAR (5) , -- CustomerType
Address CHAR (35) ,
Phone  CHAR (20) ,
Dummy1 CHAR (145),
Dummy2 CHAR (126),
-- total length adjusted to 400 bytes in Oracle
CONSTRAINT Cust_PK
       PRIMARY KEY (CNo),
CONSTRAINT Sex_CHK
       CHECK (Sex IN ('M','F','N'))
)
TABLESPACE DBTechTS
STORAGE
(initial 500M NEXT 50M MAXEXTENTS UNLIMITED) ;
timing stop
timing start
CREATE TABLE InvoicePlus (
INo   CHAR (8) NOT NULL , -- Invoice ID
CNo   CHAR (8) NOT NULL , -- Customer ID, cardinality 1 000 000
IEur  DECIMAL (11,2) NOT NULL , -- Invoiced amount, cardinality 100 000
IDate DATE NOT NULL , -- Invoicing date, cardinality 2 500
```

```
IRefd CHAR (24), -- dummy
City CHAR (20) , -- cardinality 200
-- total length about 80 bytes
CONSTRAINT InvoicePlus_PK
       PRIMARY KEY (INo),
CONSTRAINT InvPlus_IEur_CHK
       CHECK (IEur > 0)
)
TABLESPACE DBTechTS
STORAGE
(initial 400M NEXT 50M MAXEXTENTS UNLIMITED) ;
timing stop

-- At this phase the contents were loaded to the tables
-- million rows to table Cust
Elapsed time was:     00:02:30.36
CPU time was:         00:00:19.70
-- and 4 million rows to InvoicePlus
Elapsed time was:     00:13:15.03
CPU time was:         00:00:35.40

-- Adding Foreign Key
ALTER TABLE InvoicePlus
ADD CONSTRAINT InvoicePlus_Cust_FK
       FOREIGN KEY (CNo) REFERENCES Cust (CNo);
-- Elapsed time about 1 min
-- Creating indeks for the Foreign key
timing start
CREATE INDEX InvoicePlus_Cust_FK
ON InvoicePlus (CNo)
TABLESPACE DBTechTS ;
timing stop
Elapsed: 00:00:51.84

-- Eliminating external ”noice” at the beginning of every Step:
--   local server, no network traffic, no other users
--   Clearing the SGA buffers as SYS user
SQLPLUS / AS SYSDBA
ALTER SYSTEM FLUSH SHARED_POOL;

-- Connect as DBTech user

-- Step1A
timing start
CREATE INDEX Cust_X1 ON Cust (Lname,Fname)
TABLESPACE DBTechTS ;
timing stop
-- Elapsed: 00:00:24.86

-- expanding linesize for reporting Execution Plans
SET LINESIZE 160
--
SET AUTOTRACE ON
TIMING START
-- Eliminating the print of the resulting rows
select count(*)
from
( -- the actual query
  select cno, fname, address, phone -- Step 1A
  from cust
  where lname = 'Lname287' -- FF = 0,1%
  and city = 'Truro' -- FF = 0,5%
  order by fname
```

```
) ;
TIMING STOP
SET AUTOTRACE OFF


  COUNT(*)
----------
         5


Execution Plan
----------------------------------------------------------
Plan hash value: 3483299014
-------------------------------------------------------------------------------------------
| Id  | Operation                     | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |         |     1 |    39 |   929   (1)| 00:00:12 |
|   1 |  SORT AGGREGATE               |         |     1 |    39 |            |          |
|*  2 |   TABLE ACCESS BY INDEX ROWID| CUST    |   102 |  3978 |   929   (1)| 00:00:12 |
|*  3 |    INDEX RANGE SCAN           | CUST_X1 |   938 |       |     8   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter("CITY"='Truro')
   3 - access("LNAME"='Lname287')

Note
-----
   - dynamic sampling used for this statement


Statistics
----------------------------------------------------------
        488  recursive calls
          0  db block gets
       1176  consistent gets
       1282  physical reads
          0  redo size
        411  bytes sent via SQL*Net to client
        385  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          5  sorts (memory)
          0  sorts (disk)
          1  rows processed

SQL> TIMING STOP
Elapsed: 00:00:09.85
SQL> SET AUTOTRACE OFF


--- The following tests are run just for comparison to see
--  if the effect if we let the rows be printed
--  Note: this is biased, since we forgot the clear the SGA!
SET AUTOTRACE ON
TIMING START
select cno, fname, address, phone -- Step 1A
from cust
where lname = 'Lname287' -- FF = 0,1%
and city = 'Truro' -- FF = 0,5%
order by fname
;
TIMING STOP
```

```
SET AUTOTRACE OFF

CNO      FNAME           ADDRESS                              PHONE
-------- --------------- ------------------------------------ --------------------
-
00970248 Mname199        17, New Change Street                +44-123-9702480
00269282 Wname118        34, Lovat st                         +44-123-2692820
00907637 Wname147        44, Albion                           +44-123-9076370
00352496 Wname279        67, Green Arbour Highwalk            +44-123-3524960
00545959 Wname326        2, Salisbury Estate                  +44-123-5459590


Execution Plan
----------------------------------------------------------
Plan hash value: 880069320

-------------------------------------------------------------------------------------------
| Id  | Operation                    | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |         |   102 | 12750 |   930   (1)| 00:00:12 |
|   1 |  SORT ORDER BY               |         |   102 | 12750 |   930   (1)| 00:00:12 |
|*  2 |   TABLE ACCESS BY INDEX ROWID| CUST    |   102 | 12750 |   929   (1)| 00:00:12 |
|*  3 |    INDEX RANGE SCAN          | CUST_X1 |   938 |       |     8   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter("CITY"='Truro')
   3 - access("LNAME"='Lname287')

Note
-----
   - dynamic sampling used for this statement


Statistics
----------------------------------------------------------
        555  recursive calls
          0  db block gets
       1181  consistent gets
        109  physical reads
          0  redo size
       1031  bytes sent via SQL*Net to client
        385  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          6  sorts (memory)
          0  sorts (disk)
          5  rows processed

SQL> TIMING STOP
Elapsed: 00:00:00.90
SQL> SET AUTOTRACE OFF
SQL>
CNO      FNAME           ADDRESS
-------- --------------- ------------------------------------
PHONE
--------------------
00970248 Mname199        17, New Change Street
+44-123-9702480

00269282 Wname118        34, Lovat st
+44-123-2692820
```

```
00907637 Wname147        44, Albion
+44-123-9076370


CNO      FNAME           ADDRESS
-------- --------------- -----------------------------------
PHONE
--------------------
00352496 Wname279        67, Green Arbour Highwalk
+44-123-3524960

00545959 Wname326        2, Salisbury Estate
+44-123-5459590



Execution Plan
----------------------------------------------------------
Plan hash value: 880069320
-------------------------------------------------------------------------------------------
| Id  | Operation                     | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |         |     7 |   833 |  1461   (1)| 00:00:18 |
|   1 |  SORT ORDER BY                |         |     7 |   833 |  1461   (1)| 00:00:18 |
|*  2 |   TABLE ACCESS BY INDEX ROWID| CUST    |     7 |   833 |  1460   (1)| 00:00:18 |
|*  3 |    INDEX RANGE SCAN           | CUST_X1 |  1446 |       |    11   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter("CITY"='Truro')
   3 - access("LNAME"='Lname287')

Statistics
----------------------------------------------------------
        704  recursive calls
          0  db block gets
       1172  consistent gets
       1007  physical reads
          0  redo size
       1031  bytes sent via SQL*Net to client
        385  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
         29  sorts (memory)
          0  sorts (disk)
          5  rows processed

SQL> TIMING STOP
Elapsed: 00:00:07.96
SQL> SET AUTOTRACE OFF
```

Appendix 3 IDBLab2 Sample using SQL Server

.. to be included in later versions of this document

References and Links

[1] Thomas Connolly and Carolyn Begg, Database Systems, 5th ed. 2009, Addison-Wesley
[2] C. J. Date, An Introduction to Database Systems, 8th ed., Addison-Wesley, 2004
[3] Raul F. Chong et al, Understanding DB2 Learning Visually with Examples, 2nd ed, IBM Press, 2008
[4] Peter Gulutzan and Trudy Pelzer, SQL Performance Tuning, Addison-Wesley, 2003
[5] Tapio Lahdenmäki and Michael Leach, Relational Database Index Design and the Optimizers, Wiley-Interscience, 2005
[6] Craig S. Mullins, Database Administration, The complete Guide to Practices and Procedures, Addison-Wesley, 2002
[7] Raghu Ramakrishnan and Johannes Gehrke, Database Management Systems, 3rd ed. McGraw-Hill, 2003
[8] IBM, DB2 Version 9.5 for Linux, UNIX, and Windows -Tuning Database Performance, March 2008
[9] Microsoft, SQL Server 2008 Books Online
[10] Oracle® Database Performance Tuning Guide 11g Release 1, July 2008
[11] X/Open, Data Management: Structured Query Language (SQL) Version 2, 1996

Index