<u>Database Clusters:</u>

A database cluster is a collection of databases that will be accessible through a single instance of a running database server

A cluster index is needed in support of a cluster. One is used to allow the other to exist. Much like we need an index in support of a primary key. They are not the same things.

Clusters are useful in the database to store related pieces of information from more then 1 table in the same physical database block. It in effect stores data "prejoined". I can use this technique to store all of the data from the DEPT and EMP table for a given DEPTNO on the same block so that all employees of department 10 as well as the department 10 master record are all physically stored on the same exact block. When I go to "join" this data -- it is already done for me -- in a very few IOs I can get all of the data I need. Using conventional tables, this data could be scattered onto many dozens of blocks.

Clusters are useful when you want data with the same cluster key values to be physically stored near eachother.

Hash clusters are useful when you always access the data by primary key. For example, say you have some data in a table T and you ALWAYS query:

```
  select * from T where id = :x;
```

That might be a good candidate for a hash cluster since there will be no index needed. Oracle will hash the value of :x into a physical address and go right to the data. No index range scan, just a table access by the hash value.

Also, with a hash cluster there is no index by definition so no syntax to inspect. We hash the cluster key to determine where the data goes -- we do not index it. The data is the index.

Clustered and Non-Clustered Index in SQL Server

Creating a Table

To better explain SQL Server non-clustered indexes; let's start by creating a new table and populating it with some sample data using the following scripts. I assume you have a database you can use for this. If not, you will want to create one for these examples.

```
Create Table DummyTable1
(
EmpId Int,
EmpName Varchar(8000)
)
```

When you first create a new table, there is no index created by default. In technical terms, a table without an index is called a "heap". We can confirm the fact that this new table doesn't have an index by taking a look at the sysindexes system table, which contains one for this table with an of indid = 0. The sysindexes table, which exists in every database, tracks table and index information. "Indid" refers to Index ID, and is used to identify indexes. An indid of 0 means that a table does not have an index, and is stored by SQL Server as a heap.

Now let's add a few records in this table using this script:

```
Insert Into DummyTable1 Values (4, Replicate ('d',2000))
GO

Insert Into DummyTable1 Values (6, Replicate ('f',2000))
GO

Insert Into DummyTable1 Values (1, Replicate ('a',2000))
GO

Insert Into DummyTable1 Values (3, Replicate ('c',2000))
GO
```

Now, let's view the contests of the table by executing the following command in Query Analyzer for our new table.

```
Select EmpID From DummyTable1
GO
```

```
    Empid
        4
        6
        1
```

As you would expect, the data we inserted earlier has been displayed. Note that the order of the results is in the same order that I inserted them in, which is in no order at all.

Now, let's execute the following commands to display the actual page information for the table we created and is now stored in SQL Server.

dbcc ind(dbid, tabid, -1) – This is an undocumented command.

```
DBCC TRACEON (3604)
GO

Declare @DBID Int, @TableID Int
Select @DBID = db_id(), @TableID = object_id('DummyTable1')

DBCC ind(@DBID, @TableID, -1)
GO
```

This script will display many columns, but we are only interested in three of them, as shown below.

```
PagePID         IndexID     PageType
26408                 0           10
26255                 0            1
26409                 0            1
```

Here's what the information displayed means:

PagePID is the physical page numbers used to store the table. In this case, three pages are currently used to store the data.

IndexID is the type of index,

Where:

0 – Datapage

1 – Clustered Index

2 – Greater and equal to 2 is an Index page (Non-Clustered Index and ordinary index),

PageType tells you what kind of data is stored in each database,

Where:

10 – IAM (Index Allocation MAP)

1 – Datapage

2 – Index page

Now, let us execute DBCC PAGE command. This is an undocumented command.

```
DBCC page(dbid, fileno, pageno, option)
```

Where:

dbid = database id.

Fileno = fileno of the page. Usually it will be 1, unless we use more than one file for a database.

Pageno = we can take the output of the dbcc ind page no.

Option = it can be 0, 1, 2, 3. I use 3 to get a display of the data. You can try yourself for the other options.

Run this script to execute the command:

```
DBCC TRACEON (3604)
GO

DBCC page(@DBID, 1, 26408, 3)
GO
```

The output will be page allocation details.

```
DBCC TRACEON (3604)
GO

dbcc page(@DBID, 1, 26255, 3)
GO
```

The data will be displayed in the order it was entered in the table. This is how SQL stores the data in pages. Actually, 26255 & 26409 both display the data page.

I have displayed the data page information for page 26255 only. This is how MS SQL stores the contents in data pages as such column name with its respective value.

Record Type = PRIMARY_RECORD

EmpId = 4

EmpName = ddddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd

Record Type = PRIMARY_RECORD

EmpId = 6

EmpName = ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

Record Type = PRIMARY_RECORD

EmpId = 1

EmpName = aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

This displays the exact data storage in SQL, without any index on table. Now, let’s go and create a unique non-clustered index on the EmpID column.

Creating a Non-Clustered Index

Now, we will create a unique non-clustered index on the empid column to see how it affects the data, and how the data is stored in SQL Server.

```
CREATE UNIQUE NONCLUSTERED INDEX DummyTable1_empid
ON DummyTable1 (empid)
GO
```

Now, execute the DBCC ind (dbid, tabid, -1)

```
DBCC TRACEON (3604)
GO

Declare @DBID Int, @TableID Int
Select @DBID = db_id(), @TableID = object_id('DummyTable1')
DBCC ind(@DBID, @TableID, -1)
GO
```

Here are the results:

```
PagePID              IndexID       PageType
26408                      0             10
26255                      0              1
26409                      0              1
26411                      2             10
26410                      2              2
```

Now, we see two more rows than before, which now contains index page details. Page 26408 displays the page allocation details, and pages 26255 and 26409 display the data page details, as before.

In regard to the new pages, page 26411 displays the page allocation details of an index page and page 26410 displays the index page details.

MS SQL generates a page (pagetype = 10) for an index and explains the page allocation details for an index. It shows the number of index page have been occupied for an index.

Let us see what would be the output for page 26411, that is page type = 10

```
IAM: Single Page Allocations @0x308A608E

-----------------------------------------

Slot 0 = (1:26410)
```

Let us view page 26410 to see the index page details.

```
DBCC TRACEON (3604)
GO
```

```
DBCC page(10, 1, 26410, 3)
GO
```

SQL populates the index column data in order. The last column (?) is pointed to the row locator.

Here are the results, using two different methods:

Method I

FileID	PageID	EMPID	?
1	26410	1	0x8F66000001000200
1	26410	3	0x2967000001000000
1	26410	4	0x8F66000001000000
1	26410	6	0x8F66000001000100

The row location display in one of two ways:

- If the table does not have a clustered index, the row locator will be combination of fileno, pageno and the no of rows in a page.
- If the table does have clustered index, the row location will be clustered index key value.

Non-clustered indexes are particularly handy when we want to return a single row from a table.

For example, to search for employee ID (empid = 3) in a table that has a non-clustered index on the empid column, SQL Server looks through the index to find an entry that lists the exact page and row in the table where the matching empid can be found, and then goes directly to that page and row. This greatly speeds up accessing the record in question.

```
Select EmpID, EmpName From DummyTable1 WHERE EMPID = 3
```

Now, let’s insert some more rows in our table and view the data page storage of our non-clustered index.

```
Insert Into DummyTable1 Values (10, Replicate ('j',2000))
GO

Insert Into DummyTable1 Values (2, Replicate ('b',2000))
GO

Insert Into DummyTable1 Values (5, Replicate ('e',2000))
GO
```

```
Insert Into DummyTable1 Values (8, Replicate ('h',2000))
GO

Insert Into DummyTable1 Values (9, Replicate ('i',2000))
GO

Insert Into DummyTable1 Values (7, Replicate ('g',2000))
GO
```

Now, let’s view the data in our table.

Execute:

```
Select EmpID From DummyTable1
```

Here are the results:

```
EmpID
    4
    6
    1
    3
   10
    2
    5
    8
    9
    7
```

As you may notice above, the data is still in the order we entered it, and not in any particular order. This is because adding the non-clustered index didn’t change how the data was stored and ordered on the data pages.

Now, let’s view the results of the DBCC IND command. In order to find out what happened when the new data was added to the table.

```
DBCC TRACEON (3604)
GO

Declare @DBID Int, @TableID Int
Select @DBID = db_id(), @TableID = object_id('DummyTable1')
DBCC ind(@DBID, @TableID, -1)
GO
```

Here are the results:

```
PagePID             IndexID       PageType
26408                     0             10
26255                     0              1
26409                     0              1
26412                     0              1
26413                     0              1
26411                     2             10
26410                     2              2
```

Let us execute the page 26410 again and get the index page details.

```
DBCC TRACEON (3604)
GO

dbcc page(10, 1, 26410, 3)
GO
```

SQL Server populates the index column data in order. The last column (?) is pointed to the row locator.

Here are the results:

Method I

```
    FileID PageID          EMPID ?
         1 26410               1            0x8F66000001000200
         1 26410               2            0x2C67000001000000
         1 26410               3            0x2967000001000000
         1 26410               4            0x8F66000001000000
         1 26410               5            0x2C67000001000100
         1 26410               6            0x8F66000001000100
         1 26410               7            0x2D67000001000000
         1 26410               8            0x2C67000001000200
         1 26410               9            0x2967000001000200
         1 26410              10            0x2967000001000100
```

As I explained earlier, there are two types of row locations. We have seen Method I. Now, let’s try Method II with the help of a clustered and non-clustered index in a table. DummyTable1 already has a non-clustered index. Let’s now add a new column to the DummyTabl1 table and add a clustered index on that column.

```
Alter Table DummyTable1 Add EmpIndex Int IDENTITY(1,1)
GO
```

This will link the clustered index key value, instead of the row locator, and be will the combination of fileno, pageno and no of rows in a page.

This adds the Empindex column to DummyTable1. I have used an identity column so that we will not have null values on that column.

You can execute the DBCC ind and DBCC page to check if there any change after the new column is added to the table. If you don't want to check this yourself, I can tell you that adding the new column did not affect the total number of pages currently allocated to the table by SQL Server.

Now, let's add a unique clustered index on the empindex column and then view the differences in page 26410.

First, we execute the DBCC ind command. This displays a new set of pages for dummytable1.

```
DBCC TRACEON (3604)
GO

Declare @DBID Int, @TableID Int
Select @DBID = db_id(), @TableID = object_id('DummyTable1')
DBCC ind(@DBID, @TableID, -1)
GO
```

Here are the results:

```
PagePID         IndexID     PageType
26415                 1           10
26414                 0            1
26416                 1            2
26417                 0            1
26418                 0            1
26420                 2           10
26419                 2            2
```

Pages 26415 and 26420 have page allocation details. Pages 26414, 26417 and 26418 have data page details.

Now, let's view pages 26416 and 26419 and see the output.

```
DBCC TRACEON (3604)
GO

DBCC page(10, 1, 26416, 3)
GO
```

Here are the results:

FileID	PageID	ChildPageID	EMPID
1	26416	26414	0
1	26416	26417	5
1	26416	26418	9

This displays the output of the clustered index page, which has got a link to data page (ChildPageID). EMPID is an index column that contains the starting row of the page.

```
DBCC TRACEON (3604)
GO
```

```
DBCC page(10, 1, 26419, 3)
GO
```

Here are the results:

Method II

FileID	PageID	EMPID	EMPIndex
1	26419	1	1
1	26419	2	2
1	26419	3	3
1	26419	4	4
1	26419	5	5
1	26419	6	6
1	26419	7	7
1	26419	8	8
1	26419	9	9
1	26419	10	10

It is interesting to see the differences now. There is a difference between Method I and Method II. Method II is now linked to a clustered index key.

The main difference between Method I and Method II is the link to a row in a data page.

Part II: Clustered Index

Creating a Table

To better explain how SQL Server creates clustered indexes; let's start by creating a new table and populating it with some sample data using the following scripts. You can use the same sample database as before.

```
Create Table DummyTable2

(
   EmpId Int,
   EmpName Varchar(8000)
)
```

As in the previous example, when you first create a new table, there is no index created by default, and a heap is created. As before, we can confirm the fact that this new table doesn't have an index by taking a look at the sysindexes system table, which contains one for this table with an of indid = 0. The sysindexes table, which exists in every database, tracks table and index information. "Indid" refers to Index ID, and is used to identify indexes. An indid of 0 means that a table does not have an index, and is stored by SQL Server as a heap.

Now let's add a few records in this table using this script:

```
Insert Into DummyTable2 Values (4, Replicate ('d',2000))
GO

Insert Into DummyTable2 Values (6, Replicate ('f',2000))
GO

Insert Into DummyTable2 Values (1, Replicate ('a',2000))
GO

Insert Into DummyTable2 Values (3, Replicate ('c',2000))
GO
```

Now, let's view the contents of the table by executing the following command in Query Analyzer for our new table.

```
Select EmpID From DummyTable2
GO
```

```
     Empid
          4
          6
          1
          3
```

As you would expect, the data we inserted has been displayed. Note that the order of the results is in the same order that I inserted them in, which is in no order at all.

Now, let’s execute the following commands to display the actual page information for the table we created and is now stored in SQL Server.

DBCC ind(dbid, tabid, -1) – It is an undocumented command.

```
DBCC TRACEON (3604)
GO

Declare @DBID Int, @TableID Int
Select @DBID = db_id(), @TableID = object_id('DummyTable2')
DBCC ind(@DBID, @TableID, -1)
GO
```

This script will display many columns, but we are only interested in three of them, as shown below.

Here are the results:

```
PagePID          IndexID     PageType
26408                  0           10
26255                  0            1
26409                  0            1
```

Here’s what the information displayed means:

PagePID is the physical page numbers used to store the table. In this case, three pages are currently used to store the data.

IndexID is the type of index,

Where:

0 – Datapage

1 – Clustered Index

2 – Greater and equal to 2 is an Index page (Non-Clustered Index and ordinary index)

PageType tells you what kind of data is stored in each database

Where:

10 – IAM (Index Allocation MAP)

1 – Datapage

2 – Index page

Now, let us execute DBCC PAGE command.

```
DBCC page(dbid, fileno, pageno, option)
```

Where:

dbid = database id.

Fileno = fileno of the page. Usually it will be 1, unless we use more than one file for a database.

Pageno = we can take the output of the dbcc ind page no.

Option = it can be 0, 1, 2, 3. I use 3 to get a display of the data. You can try yourself for the other options.

Run this script to execute the command:

```
DBCC TRACEON (3604)
GO

DBCC page(@DBID, 1, 26408, 3)
GO
```

The output will be page allocation details.

```
DBCC TRACEON (3604)
GO

DBCC page(@DBID, 1, 26255, 3)
GO
```

The output will display the data however it was entered in the table. This is how SQL stores the data in pages. Actually, 26255 & 26409 will display the data page.

I have displayed the data page information for page 26255 only. This is how MS-SQL stores the contents in data pages as such column name with its respective value.

```
Record Type = PRIMARY_RECORD
```

EmpId = 4

EmpName = dddddddddddddddddddddddddddddddddddddddddddddddd
dddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
dddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
dddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
dddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd

Record Type = PRIMARY_RECORD

EmpId = 6

EmpName = ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

Record Type = PRIMARY_RECORD

EmpId = 1

EmpName = aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

This displays the exact data storage in SQL without any index on table. Now, let’s go and create a Unique Clustered Index on EmpID column.

Create a Clustered Index

Now, let us create a unique clustered index on empid column to see how it affects the data that is stored in SQL Server.

```
CREATE UNIQUE CLUSTERED INDEX DummyTable2_EmpIndex
ON DummyTable2 (EmpID)
GO
```

Execute:

```
Select EmpID From DummyTable2
```

Here are the results:

```
    Empid
        1
        3
        4
        6
```

Now, execute the DBCC ind (dbid, tabid, -1)

```
DBCC TRACEON (3604)
GO

Declare @DBID Int, @TableID Int

Select @DBID = db_id(), @TableID = object_id('DummyTable2')

DBCC ind(@DBID, @TableID, -1)
GO
```

Here are the results:

```
PagePID          IndexID      PageType
26411                  1            10
26410                  0             1
26412                  1             2
```

MS SQL generates a page (pagetype = 10) for an index and explains the page allocation details for an index. It shows the number of index page have been occupied for an index.

Now, let us view the page 26410 and 26412 and see the page details.

```
DBCC TRACEON (3604)
GO

DBCC page(10, 1, 26412, 3)
GO
```

Here are the results:

```
    FileID       PageID       ChildPageID EMPID
         1        26412            26410 0
```

The output display many columns, but we are only interested in four of them as shown above.

This will display the output of the index page, which has got link to data page (ChildPageID). EMPID is an index column will contain the starting row of the page.

Now, let us view the page 26410 and see the page details.

```
DBCC TRACEON (3604)
GO

DBCC page (10, 1, 26410, 3)
GO
```

Here are the results:

```
Record Type = PRIMARY_RECORD

EmpId           = 1

EmpName         = aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa



Record Type = PRIMARY_RECORD

EmpId           = 2

EmpName         = bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb



Record Type = PRIMARY_RECORD

EmpId           = 3
```

```
EmpName          = cccccccccccccccccccccccccccccccccccccccccccccc
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
```

Though I have added disorder records, SQL has displayed the data page in sequence because we have got a clustered index on empid. This is absolutely great! Adding a clustered index to the table has physically reordered the data pages, putting them in physical order based on the indexed column.

Now, let’s insert some more rows in our table and view the data and index page storage of our clustered index.

```
Insert Into DummyTable2 Values (10, Replicate ('j',2000))
GO

Insert Into DummyTable2 Values (2, Replicate ('b',2000))
GO

Insert Into DummyTable2 Values (5, Replicate ('e',2000))
GO

Insert Into DummyTable2 Values (8, Replicate ('h',2000))
GO

Insert Into DummyTable2 Values (9, Replicate ('i',2000))
GO

Insert Into DummyTable2 Values (7, Replicate ('g',2000))
GO
```

Now, execute the DBCC ind (dbid, tabid, -1)

```
DBCC TRACEON (3604)
GO

Declare @DBID Int, @TableID Int

Select @DBID = db_id(), @TableID = object_id('DummyTable2')

DBCC ind(@DBID, @TableID, -1)
GO
```

Here are the results:

```
PagePID          IndexID     PageType
26411                  1           10
26410                  0            1
26412                  1            2
26255                  0            1
26408                  0            1
26409                  0            1
```

Now, we see few more rows than before. Page 26411 displays the page allocation details, and pages 26408, 26409, 26410 and 26255 display the data page details, as before.

In regard to the new pages, page 26411 displays the page allocation details of an index page and 26412 displays the index page details.

MS-SQL generates a page (pagetype = 10) for an index and explains the page allocation details for an index. It shows the number of index page have been occupied for an index.

Let us see what would be the output for page 26411, that is page type = 10.

```
DBCC TRACEON (3604)
GO

dbcc page(10, 1, 26411, 3)
GO
```

Here are the results:

```
IAM: Single Page Allocations @0x30A5C08E

----------------------------------------

Slot 0 = (1:26410)

Slot 1 = (1:26412)

Slot 2 = (1:26255)

Slot 3 = (1:26408)

Slot 4 = (1:26409)
```

Let us view page 26412 to see the index page details.

```
DBCC TRACEON (3604)
GO

DBCC page(10, 1, 26412, 3)
GO
```

Here are the results:

```
FileID      PageID      ChildPageID         EMPID
1           26412       26410               0
1           26412       26408               4
1           26412       26255               6
1           26412       26409               9
```

This helps us to get an idea to decide the need of clustered index. It is really useful to have a clustered index when retrieve many rows of data, ranges of data, and when BETWEEN is used in the WHERE clause. Because, the leaf level of the clustered index is the data. It should be used to save many I/Os. So, it is better to use clustered indexes to solve queries asking for ranges of data, not one row.

For example, to search for an employee ID (empid between 3 and 9) in a table that has a clustered index on the empid column.

```
Select EmpID, EmpName From DummyTable1 WHERE EMPID Between 3 And 9
```

MySQL Cluster:

First we're seeing an overview of the NDB architecture. If you've never seen it before, think "Oracle RAC without shared storage" and you're 95% of the way there.

The core NDB engine is a new storage engine inside MySQL. It provides transactions, replication, on-line backups, crash recovery, hash and tree indexes, on-line index builds, auto-detection of a failed node and re-sync when it comes back up. There are rolling upgrades, which provide a way to upgrade things without a disruption of service.

Man, this would be a lot easier if I could draw ASCII art even half as fast as I can type. Oh, well.

The NDB code is in the MySQL 4.1 tree as of today.

History. During 1991-1996, the initial design, prototypes, and research were on-going. Most of the code today originates from 1996. First demo in 1997, and it just got better from there. 1.0 came out in April 2001 w/Perl DBI support. In 1.4 (2002), node recovery was completed. 2003 brought ODBC, on-line backup, unique indexes, and more. The most recent stuff are on-ling upgrades, ordered indexes, and MySQL integration.

Benchmarks on a 72 CPU SunFire box hit 380,000 write txns/sec and 1,500,000 read txns/sec. Nice! More numbers in the slides, but I can't type that fast.

The management server handles the configuration of the cluster (config file, commands (start backup), and so on). There's a C API to the management server. It'll be integrated into MySQL at some point. MySQL uses the NDB API to talk to the cluster for normal data operations. Even with MySQL as the front end to NDB, it's still possible to use the NDB API natively to talk to the cluster too.

The config file specifies transaction timeouts, buffer sizes, heartbeat timeouts, nodes involved, and so on. Messaging diagram I can't replicate. Same with diagram of hash-based table partitioning. On-line backup and restore diagram too.

Data access methods (low-level): primary key, full table scan, unique key, parallel range scan. MySQL hides this from us, but it's interesting anyway. If you're using the NDB API, you need to know this too.

There is always a primary key. Like BDB and InnoDB, it'll create one for you if you don't specify one. Unique indexes are hash indexes, implemented as a second table (funky). It looks like unique indexes are expensive in NDB sort of like the way that secondary indexes are in InnoDB (again, I can't replicate the diagram here).

Currently working on: remaining MySQL functionality in the engine, on-line create/drop index, cluster management from the MySQL API, MySQL replication for MySQL

Cluster, on-line add/drop column. Row length limit is current ~8KB but that will change in the future.

Table and Index Structures

MySQL stores its data dictionary information of tables in `` `.frm' `` files in database directories. But every InnoDB type table also has its own entry in InnoDB internal data dictionaries inside the tablespace. When MySQL drops a table or a database, it has to delete both a `` `.frm' `` file or files, and the corresponding entries inside the InnoDB data dictionary. This is the reason why you cannot move InnoDB tables between databases simply by moving the `` `.frm' `` files, and why `DROP DATABASE` did not work for InnoDB type tables in MySQL versions <= 3.23.43.

Every InnoDB table has a special index called the clustered index where the data of the rows is stored. If you define a `PRIMARY KEY` on your table, then the index of the primary key will be the clustered index.

If you do not define a primary key for your table, InnoDB will internally generate a clustered index where the rows are ordered by the row id InnoDB assigns to the rows in such a table. The row id is a 6-byte field which monotonically increases as new rows are inserted. Thus the rows ordered by the row id will be physically in the insertion order.

Accessing a row through the clustered index is fast, because the row data will be on the same page where the index search leads us. In many databases the data is traditionally stored on a different page from the index record. If a table is large, the clustered index architecture often saves a disk I/O when compared to the traditional solution.

The records in non-clustered indexes (we also call them secondary indexes), in InnoDB contain the primary key value for the row. InnoDB uses this primary key value to search for the row from the clustered index. Note that if the primary key is long, the secondary indexes will use more space.

Physical Structure of an Index

All indexes in InnoDB are B-trees where the index records are stored in the leaf pages of the tree. The default size of an index page is 16 KB. When new records are inserted, InnoDB tries to leave 1 / 16 of the page free for future insertions and updates of the index records.

If index records are inserted in a sequential (ascending or descending) order, the resulting index pages will be about 15/16 full. If records are inserted in a random order, then the pages will be 1/2 - 15/16 full. If the fillfactor of an index page drops below 1/2, InnoDB will try to contract the index tree to free the page.

Insert Buffering

It is a common situation in a database application that the primary key is a unique identifier and new rows are inserted in the ascending order of the primary key. Thus the insertions to the clustered index do not require random reads from a disk.

On the other hand, secondary indexes are usually non-unique and insertions happen in a relatively random order into secondary indexes. This would cause a lot of random disk I/Os without a special mechanism used in InnoDB.

If an index record should be inserted to a non-unique secondary index, InnoDB checks if the secondary index page is already in the buffer pool. If that is the case, InnoDB will do the insertion directly to the index page. But, if the index page is not found from the buffer pool, InnoDB inserts the record to a special insert buffer structure. The insert buffer is kept so small that it entirely fits in the buffer pool, and insertions can be made to it very fast.

The insert buffer is periodically merged to the secondary index trees in the database. Often we can merge several insertions on the same page in of the index tree, and hence save disk I/Os. It has been measured that the insert buffer can speed up insertions to a table up to 15 times.

Adaptive Hash Indexes

If a database fits almost entirely in main memory, then the fastest way to perform queries on it is to use hash indexes. InnoDB has an automatic mechanism which monitors index searches made to the indexes defined for a table, and if InnoDB notices that queries could benefit from building of a hash index, such an index is automatically built.

But note that the hash index is always built based on an existing B-tree index on the table. InnoDB can build a hash index on a prefix of any length of the key defined for the B-tree, depending on what search pattern InnoDB observes on the B-tree index. A hash index can be partial: it is not required that the whole B-tree index is cached in the buffer pool. InnoDB will build hash indexes on demand to those pages of the index which are often accessed.

In a sense, through the adaptive hash index mechanism InnoDB adapts itself to ample main memory, coming closer to the architecture of main memory databases.

Physical Record Structure

- Each index record in InnoDB contains a header of 6 bytes. The header is used to link consecutive records together, and also in the row level locking.
- Records in the clustered index contain fields for all user-defined columns. In addition, there is a 6-byte field for the transaction id and a 7-byte field for the roll pointer.

- If the user has not defined a primary key for a table, then each clustered index record contains also a 6-byte row id field.
- Each secondary index record contains also all the fields defined for the clustered index key.
- A record contains also a pointer to each field of the record. If the total length of the fields in a record is < 128 bytes, then the pointer is 1 byte, else 2 bytes.

How an Auto-increment Column Works in InnoDB

After a database startup, when a user first does an insert to a table `T` where an auto-increment column has been defined, and the user does not provide an explicit value for the column, then InnoDB executes `SELECT MAX(auto-inc-column) FROM T`, and assigns that value incremented by one to the column and the auto-increment counter of the table. We say that the auto-increment counter for table `T` has been initialized.

InnoDB follows the same procedure in initializing the auto-increment counter for a freshly created table.

Note that if the user specifies in an insert the value 0 to the auto-increment column, then InnoDB treats the row like the value would not have been specified.

After the auto-increment counter has been initialized, if a user inserts a row where he explicitly specifies the column value, and the value is bigger than the current counter value, then the counter is set to the specified column value. If the user does not explicitly specify a value, then InnoDB increments the counter by one and assigns its new value to the column.

The auto-increment mechanism, when assigning values from the counter, bypasses locking and transaction handling. Therefore you may also get gaps in the number sequence if you roll back transactions which have got numbers from the counter.

The behavior of auto-increment is not defined if a user gives a negative value to the column or if the value becomes bigger than the maximum integer that can be stored in the specified integer type.