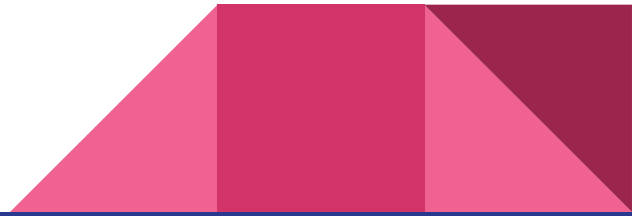
Gaining insight on RDBMS Performance

Preface

- I am not an SQL expert
- Avoid premature optimization
- Profile your queries
- I assume everyone has some basic knowledge

Won’t Cover

- ORM
- NoSQL
- Query Hints
- Columnstore
- Datawarehousing techniques
- Spatial
- Statistics
- In-memory tables
- In-memory databases
- A lot more things



Goals

- Gain insight on how data and data structures affect your query performance
- Gain insight on how the engine finds your data
- ~~Gain insight on how transactions and locking influence your performance~~

Agenda

1. Basic Techniques
2. Predicate Guidelines
3. Indexes
4. How data and database storage affects queries
5. ~~Transactions and Locking~~
6. Partitioning
7. ~~Tools~~
8. ~~Statistics~~

Basic / Misc Techniques

Basic/Misc Techniques

Minimize the data you need

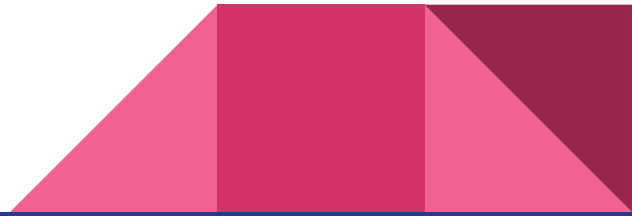
SELECT * FROM …

vs.

SELECT id, name FROM …

Negative consequences:

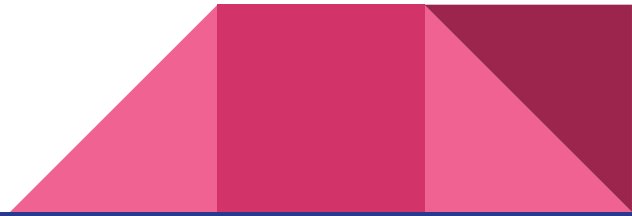
- Covering index left unused
- More processing of data
- More data sent across the network



Basic/Misc Techniques

Reduce total network latency

- Use Limit on the server side instead of allowing the client (your app) to do it
  - Ex: SELECT * FROM … LIMIT 100
- Filter as much data as possible on the server / Filter in your subqueries -> Less data sent across network



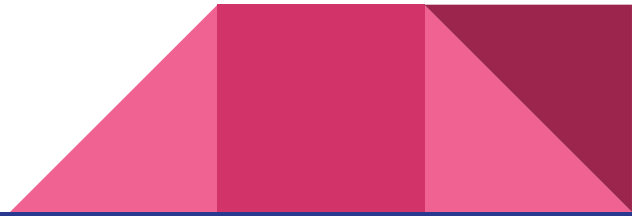
Basic/Misc Techniques

Reduce total network latency

- Question: What are the semicolons for in the following example? (DEMO)

  INSERT INTO testdb.test VALUES (1);

  INSERT INTO testdb.test VALUES (1);



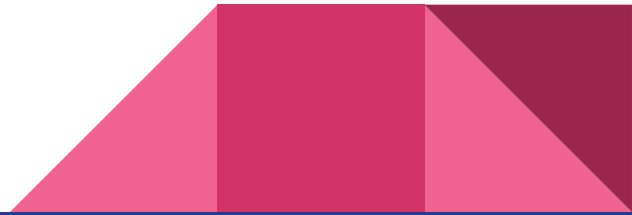
Basic/Misc Techniques

Reduce total network latency

- Send batches of statements to reduce number of round-trips
  - MySQL:

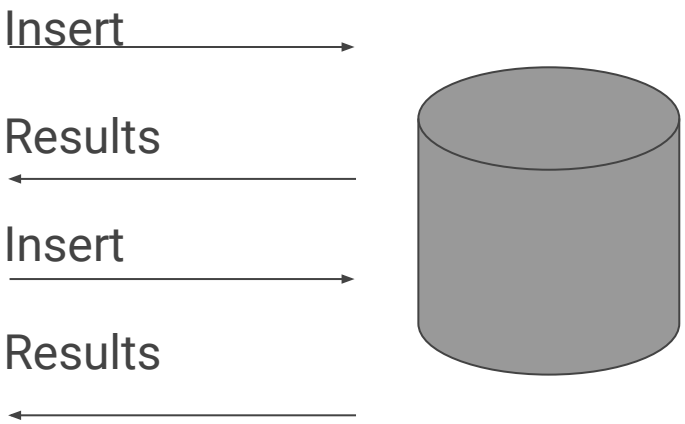
```
DELIMITER $$ -- defines a delimiter of your choice

INSERT INTO testdb.test VALUES (1);

INSERT INTO testdb.test VALUES (1);

$$
```

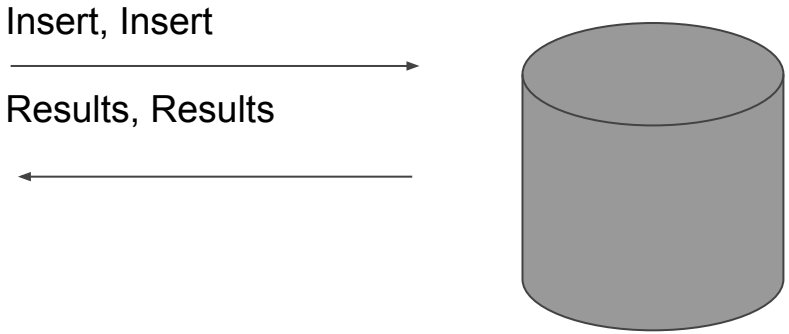
  - SQL Server uses “GO”



Basic/Misc Techniques

Reduce total network latency - batch of statements





Basic/Misc Techniques

Reduce total network latency and Reuse Resources to eliminate costs

- Connection Pooling
  - Keep connections open to reuse resources
  - Eliminates construction and destruction costs of connections

Basic/Misc Techniques

Reuse resources to eliminate costs

- Statement vs. PreparedStatement
  - RDBMS engine must create a new data execution plan for statement
  - For preparedstatements, engine can reuse the previously compiled plan

```
String updateString =  "update dbName.coffee set SALES = ? where coffee_name = ?";
updateSales = con.prepareStatement(updateString);
updateSales.setInt(1, 100);
updateSales.setString(2, “Cafe Sua Da”);
updateSales.executeUpdate();
updateSales.setString(2, "Cafe Den");
updateSales.executeUpdate();
```

Sidenote: PreparedStatements will help prevent some sql injection attacks

Sargability - Predicate Guidelines

Predicate (‘WHERE …’) Guidelines

Queries must be ***sargable*** meaning the query engine can optimize the execution plan that the query uses.

Search **ARG**ument **ABLE**

Index Seek: Can use the B+tree to fetch the matching record

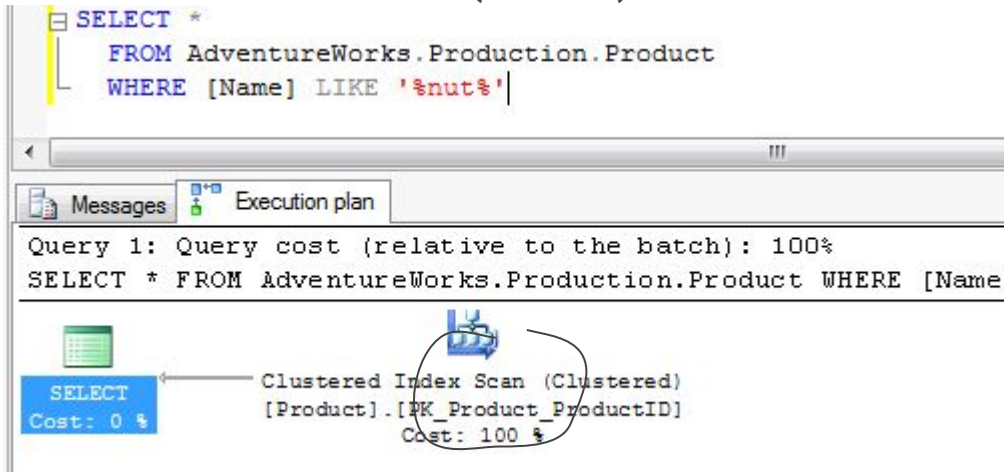
Index scan: Scans/reads all the records of the table to return the required rows.

Sargable Operators: =, >, <, >=, <=, BETWEEN, LIKE, IS [NOT] NULL, EXISTS

Predicate Guidelines

Don’t use **LIKE ‘%foo’**

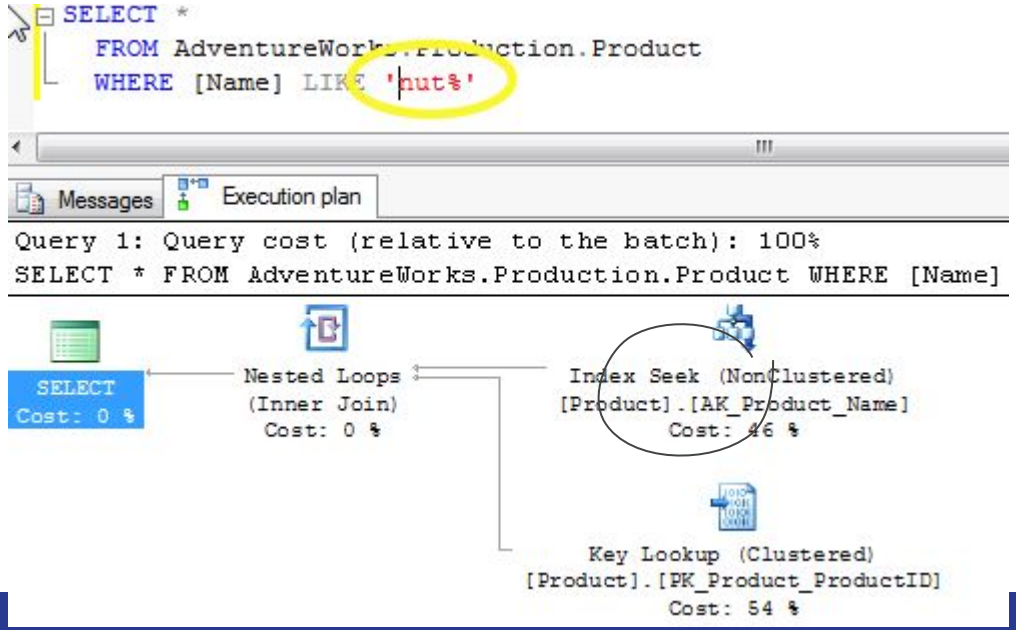
- Cannot use the index effectively. Afoo? Aafoo? Bfoo? … Zfoo? Zzfoo?
- Causes an index scan (DEMO)



Predicate Guidelines

Instead, try to use constant strings first then wildcards after LIKE ‘foo%’

- Allows index seek



Predicate Guidelines

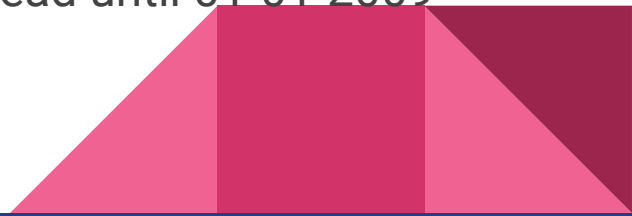
Don’t use functions in the predicate. Query engine cannot know what the output of the function will be, so it must look at every row’s output. Engine must scan.

```
WHERE Year(myDate) = 2008
```

Attempt to rewrite a query to eliminate the function.

```
WHERE myDate >= '01-01-2008' AND myDate < '01-01-2009'
```

Engine can now seek to index position ‘01-01-2008’ and read until 01-01-2009



Predicate Guidelines

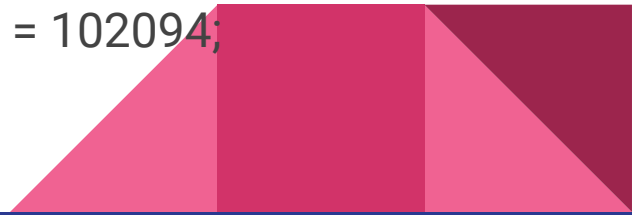
Don’t do calculations on an index column in the WHERE clause

```
SELECT Salary FROM EmployeeTest WHERE Salary / 2 = 50147 ;
```

This causes calculation on every row

Instead leave the index column values alone so we can optimize and seek (DEMO)

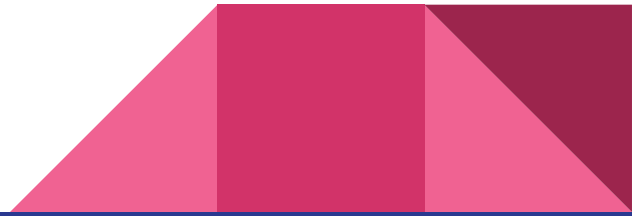
```
SELECT Salary FROM  EmployeeTest WHERE Salary  = (50147 * 2);

SELECT Salary FROM  EmployeeTest WHERE Salary  = 102094;
```



Predicate Guidelines

Don’t have implicit conversions in the predicate (DEMO)

EX: select * from test.predicateexample where id = 10.0;



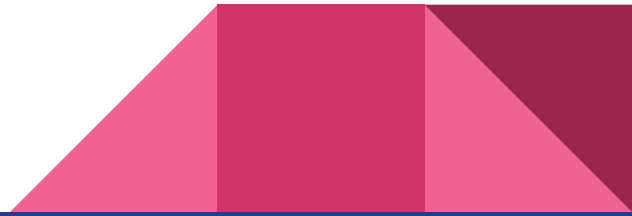
Predicate Guidelines

One more example because it’s a common pattern:

```
SELECT EmployeeName FROM  EmployeeTest WHERE ISNULL
(EmployeeName, 'Vru') = 'Vru' ;
```

Change to…

```
SELECT EmployeeName FROM  EmployeeTest WHERE EmployeeName = 'Vru'
OR EmployeeName IS NULL;
```



Indexes

Indexes

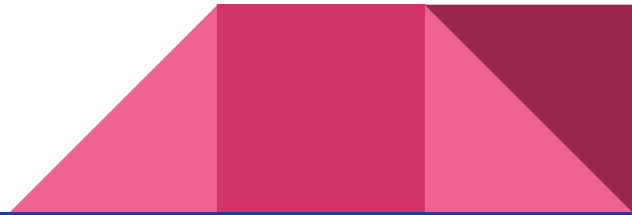
Definition:

A database index is a data structure that improves the speed of data retrieval operations on a database table at the cost of additional writes and storage space to maintain the index data structure.

Data and Database Storage

3 Main Index Types:

1. Heap - unsorted, base index
2. Clustered Index - Each row is stored in sorted order, base index
3. Non-clustered Index - Index is stored on a separate page, which just contains the key columns and a pointer into the base index.



Important Mention

A “Page” is one of the most fundamental units of data in the RDBMS.

- InnoDB: 16KB
- SQLServer: 8KB

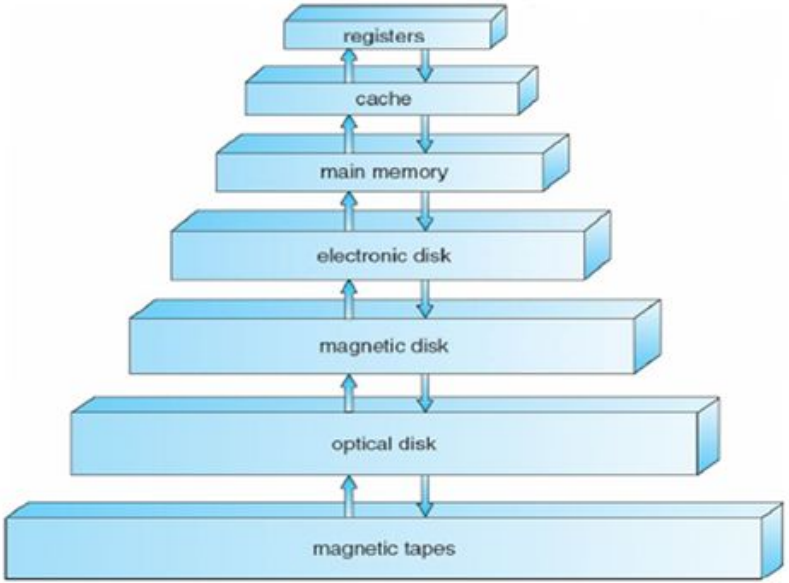


Figure 1.4 Storage-device hierarchy.

Important Mention

General Principle: Minimize the amount of IO you must do.

- Minimize the number of pages you store
- Maximize the amount data that fits into a page
- Minimize the number of pages you need for a query
- Work with contiguous Data

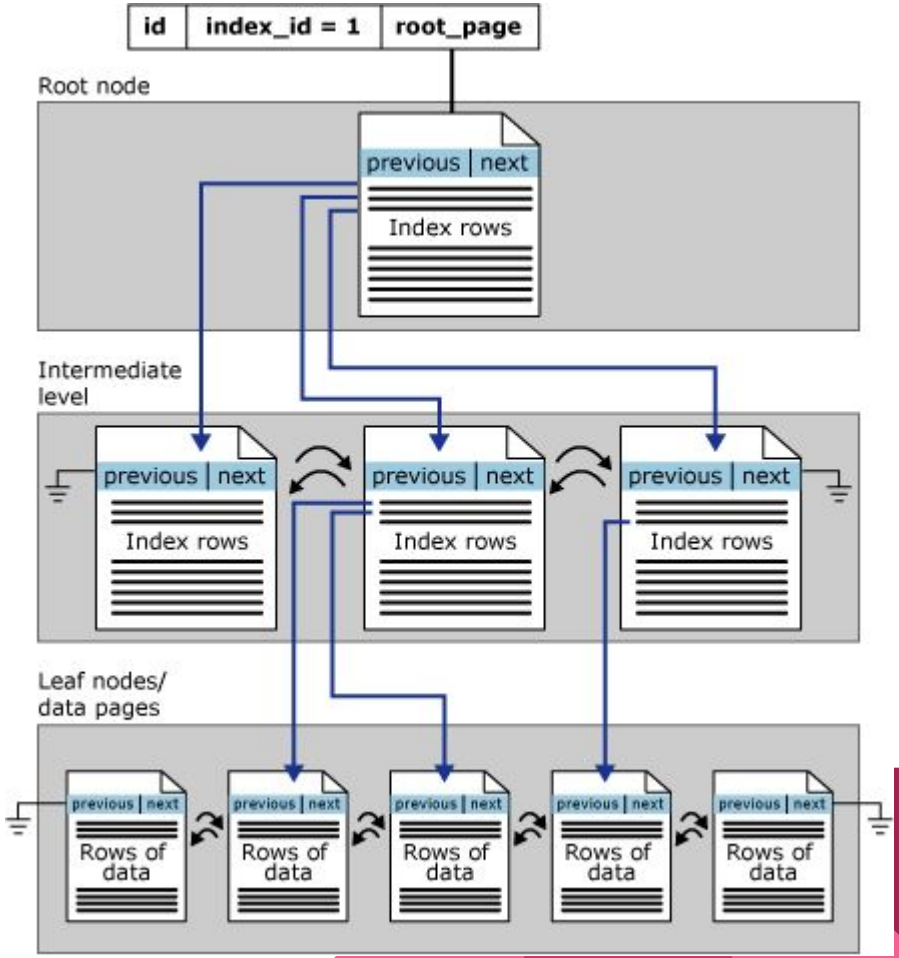
Clustered Index

What does it look like?

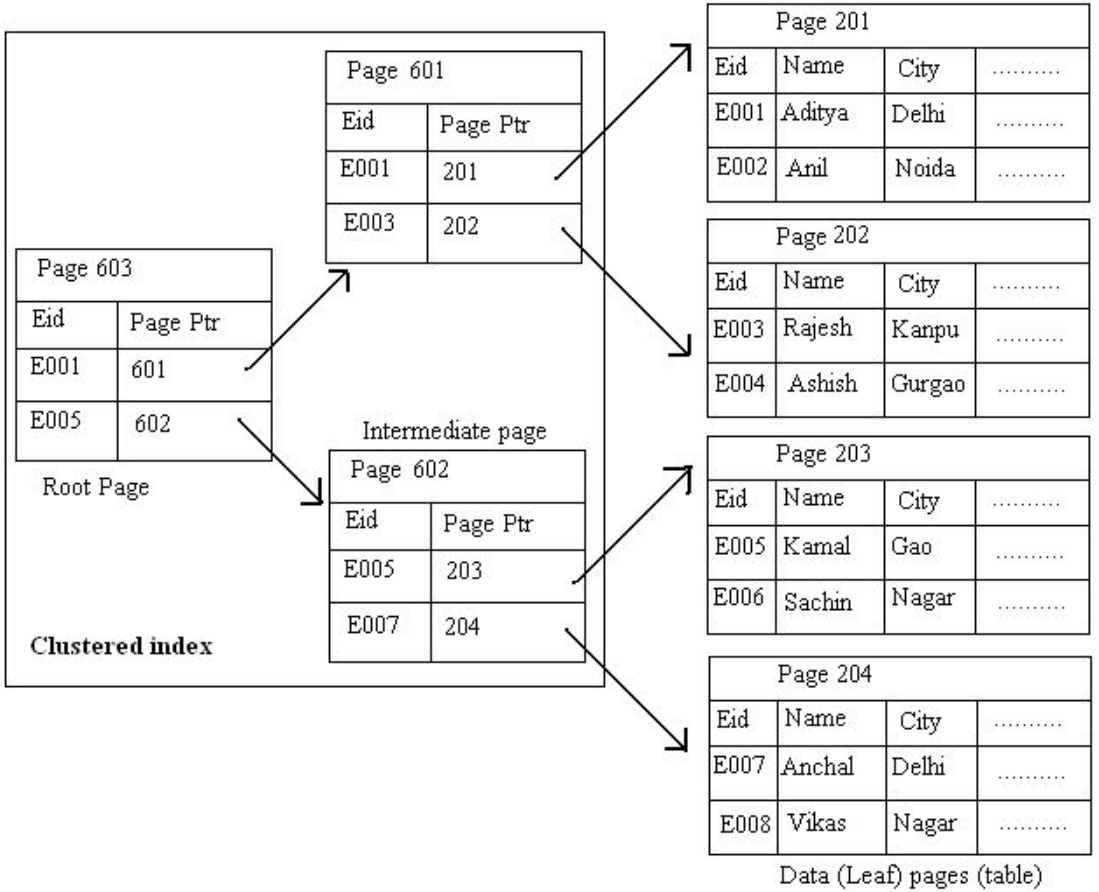
Ex: What it look like?

CREATE TABLE test.collection (id INT PRIMARY KEY, a INT, b INT, c INT);

CREATE INDEX noncls_idx on test.collection (a);

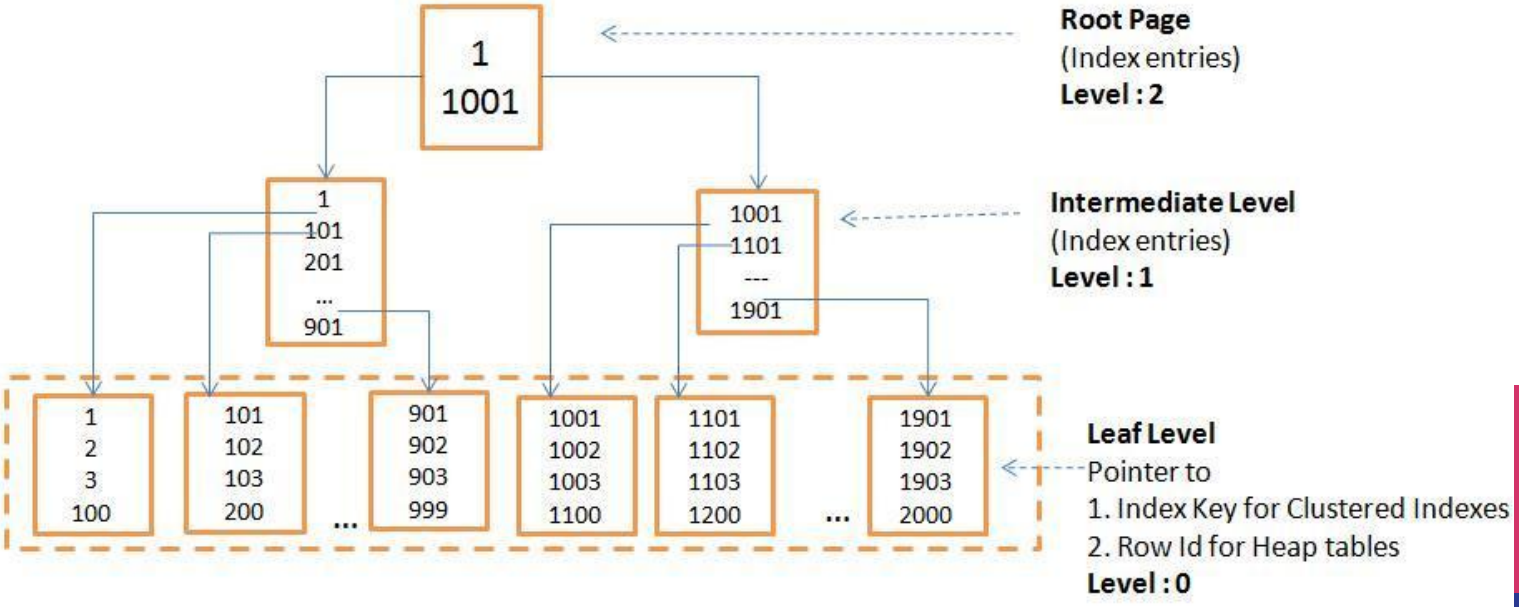


Clustered Index



Non-clustered Index

Non Clustered Index Architecture:



Data and Database Storage

General indexing guidelines:

- Index on predicate fields
- Delete unused indexes
  - If #Columns ~= ~#KeyColumns, you've almost doubled your storage
- Create a covering index
- Index on selected fields
- Index on filtered and sorted fields

Data and Database Storage

When not to use an index:

- Small Tables - Data fits on a small amount of pages. Creating an index will likely double the amount of pages required.
- Small Tables - Low cardinality will force optimizer to do a table scan, not even utilizing the index
- No queries can use your index -> Remove your index, it takes up space and requires maintenance.

Data and Database Storage

Covering Index

- A covering index is an index that contains all, and possibly more, of the columns you need for your query so no additional reads are required to get the data.
- Don’t need a base index lookup and saves I/O
- DEMO

Is my query using an index?

MySQL:

explain select * from test.dateex where id > 500 and id < 600; -- no index on id

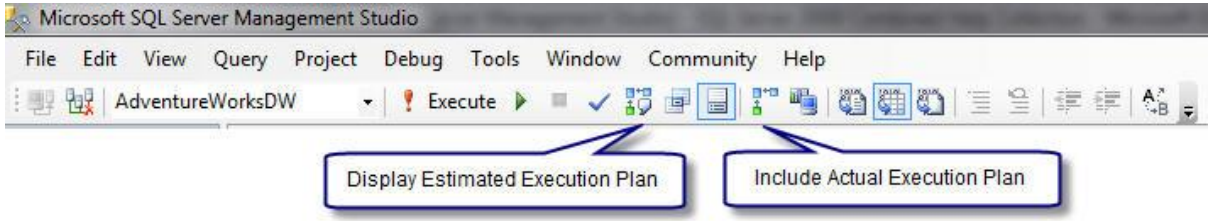
	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
▶	1	SIMPLE	dateex	NULL	ALL	NULL	NULL	NULL	NULL	2412521	11.11	Using where

explain SELECT * FROM test.dateex WHERE d BETWEEN CAST('2016-12-19 14:45:00' as DATETIME) and CAST('2016-12-19 14:45:59' as DATETIME); -- has an index on column d

	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
▶	1	SIMPLE	dateex	NULL	range	dateidx	dateidx	6	NULL	275998	100.00	Using index condition; Using MRR

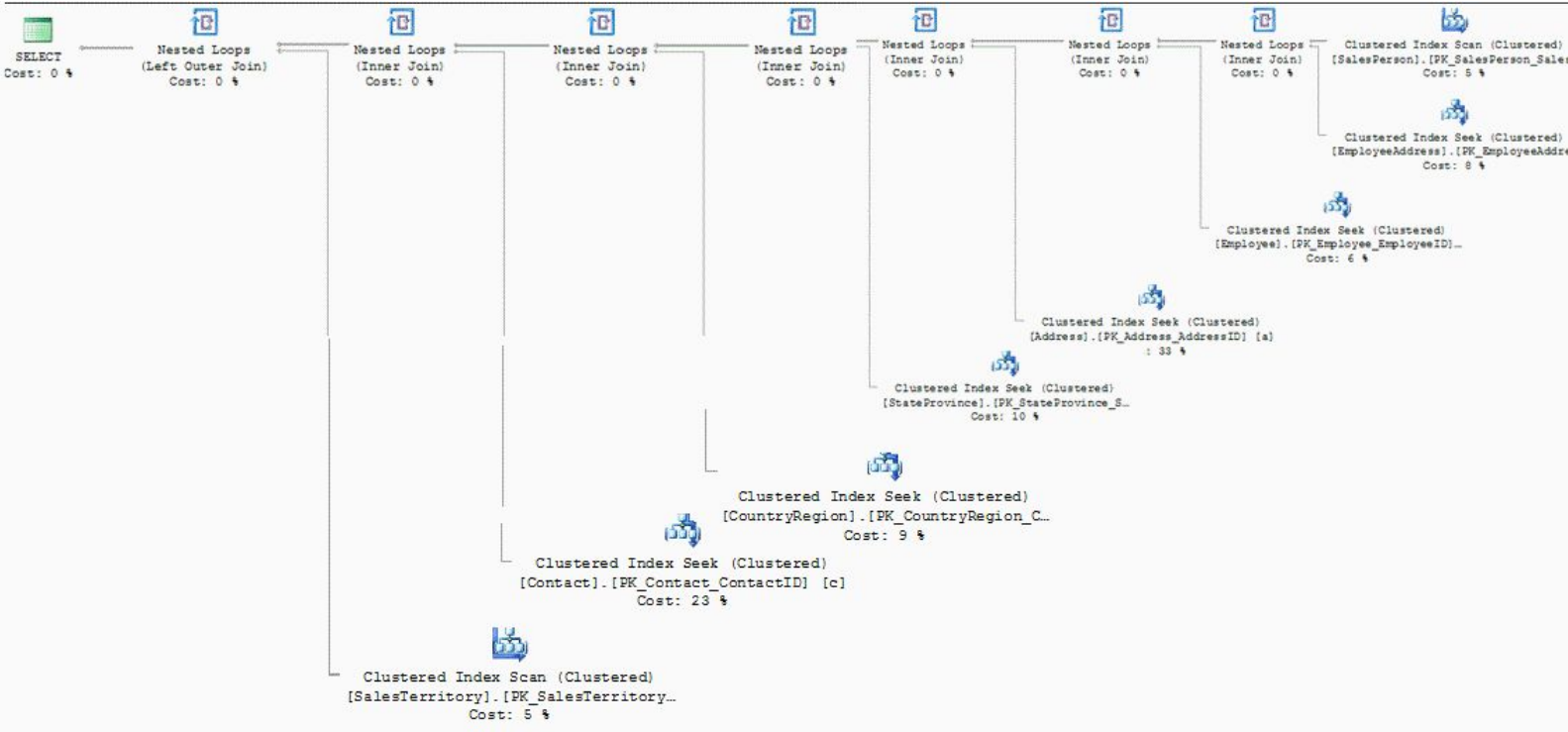
Is my query using an index?

SQL Server



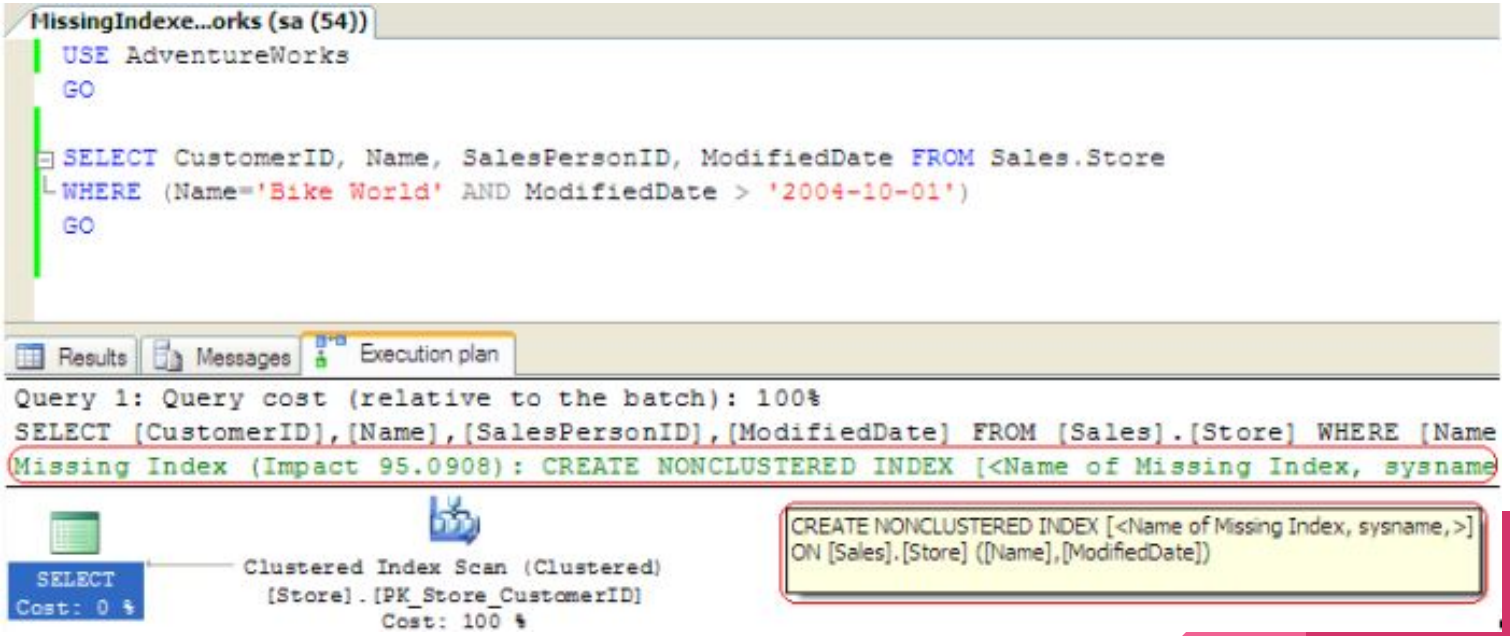
Is my query using an index?

SQL Server Management Studio



Help me create an index

SQL Server Management Studio



OR use: Select * from sys.dm_db_missing_index_details

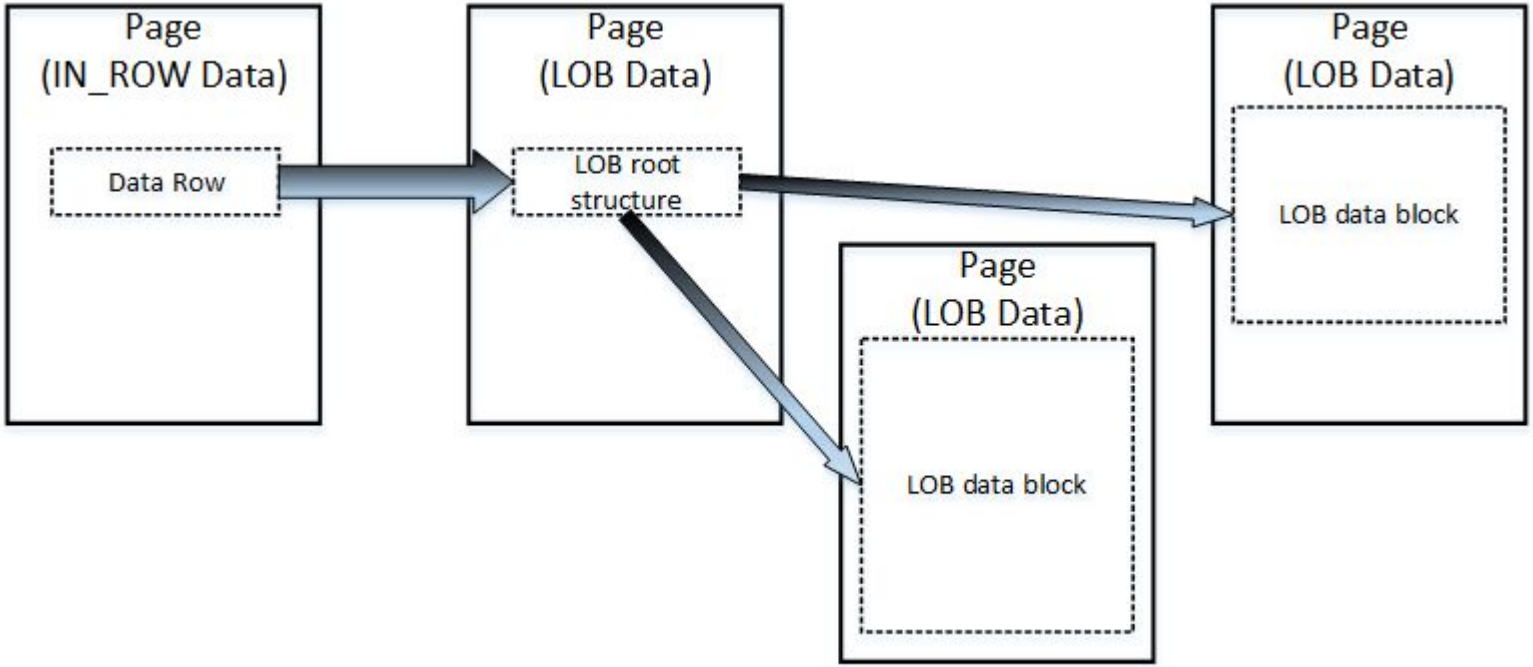
Database Pages and Page Types

Data and Database Storage

Several page types that we should be concerned with:

1. Data
2. Index
3. Row-overflow (SLOB)
    a. If a row with var length data is modified to cause the row size to become too large to fit on a page, the data is truncated and placed on a row-overflow page
    b. Row is defined to have multiple columns whose combined sizes add up to larger than a page (e.g., two 4000 byte varchar columns)
    c. Can lead to extra fragmentation, non-contiguous
4. Large object (LOB)
    a. Binary and text data types (BLOB, BINARY, XML, CLOB, TEXT, IMG)

LOB Layout


Page
(IN_ROW Data)
Data Row
Page
(LOB Data)
LOB root
structure
Page
(LOB Data)
LOB data block
Page
(LOB Data)
LOB data block

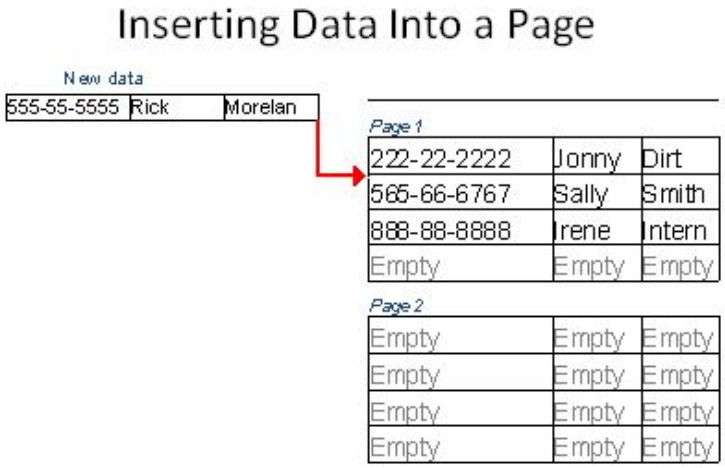
Important Mention

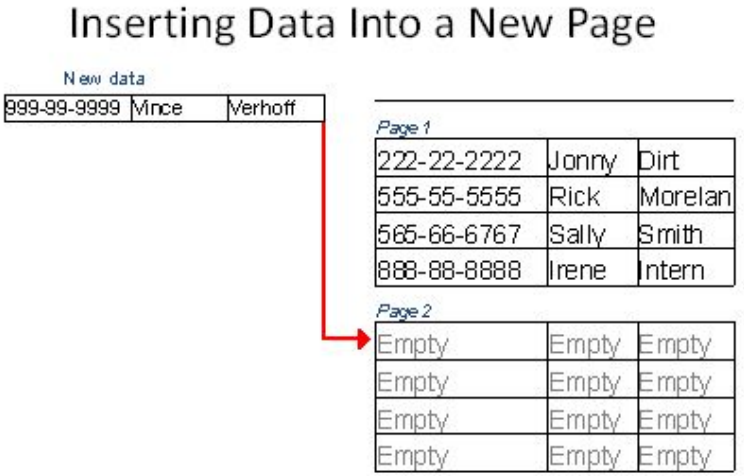
General Principle: Minimize the amount of IO you must do.

- Minimize the number of pages you store
- Maximize the amount data that fits into a page
- Minimize the number of pages you need for a query
- Work with contiguous Data

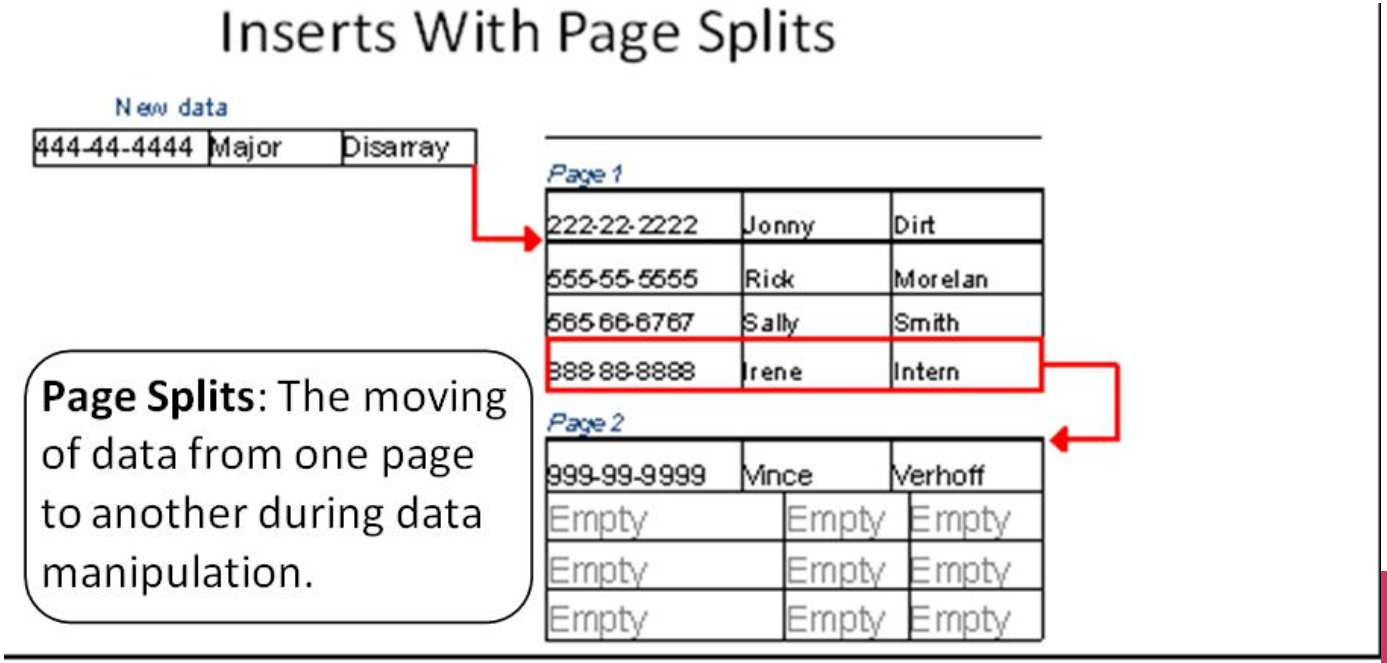
Page Splits

Page Splits: Row doesn't fit on a page, so page must be added, data must be moved into the new page. Basically I/O occurs.





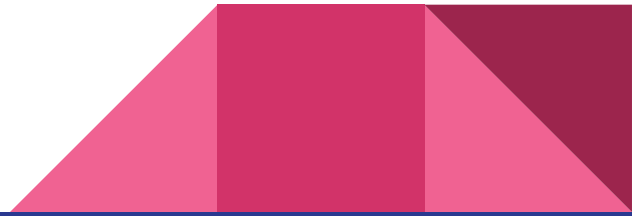
Page Splits


Inserts With Page Splits
New data
444-44-4444
Major
Disarray
Page 1
222-22-2222
Jonny
Dirt
555-55-5555
Rick
Morelan
565-66-6767
Sally
Smith
888-88-8888
Irene
Intern
Page 2
999-99-9999
Vince
Verhoff
Empty
Empty
Empty
Empty
Empty
Empty
Empty
Empty
Empty
Page Splits: The moving of data from one page to another during data manipulation.

Prevent Merge and Splits of Index Pages

“Fill Factor” and “Percent Free” settings

- In general, if your data is mostly read-only, then you can accept a 100% fill factor (maximum use of your pages). If it is read-write and can change frequently, use something smaller. ~70% is recommended, but profile your queries.
- Insert in sorted order



Fix an index with many splits with heavy defragmentation

Diagnose: SQL Server: **select ***

from sys.dm_db_index_physical_stats

Fix: For fragmentation > 30%, it is recommended to do an index rebuild will reclaim wasted disk space based on the fill factor setting, and reorder index rows into contiguous pages.

```
SELECT OBJECT_NAME(object_id), index_id,index_type_desc,index_level,
avg_fragmentation_in_percent,avg_page_space_used_in_percent,page_count
FROM sys.dm_db_index_physical_stats
(DB_ID(N'AdventureWorksLT'), null, NULL, NULL , 'SAMPLED')
ORDER BY avg_fragmentation_in_percent DESC
```

Results | Messages

	(No column name)	index_id	index_type_desc	index_level	avg_fragmentation_in_perc
1	DF_BillOfMaterials_StartDate	1	CLUSTERED INDEX	0	99.009900990099
2	DF_Address_ModifiedDate	1	CLUSTERED INDEX	0	97.0588235294118
3	DF_Contact_rowguid	1	CLUSTERED INDEX	0	94.4444444444444
4	CK_Product_SafetyStockLevel	1	CLUSTERED INDEX	0	88.5714285714286
5	ContactCreditCard	1	CLUSTERED INDEX	0	85.7142857142857
6	ContactCreditCard	256000	PRIMARY XML INDEX	0	80
7	AWBuildVersion	1	CLUSTERED INDEX	0	75
8	DF_Address_ModifiedDate	2	NONCLUSTERED INDEX	0	66.6666666666667
9	DF_Contact_rowguid	2	NONCLUSTERED INDEX	0	66.6666666666667
10	CountryRegion	1	CLUSTERED INDEX	0	60
11	CountryRegion	2	NONCLUSTERED INDEX	0	50

Fix an index with many splits with heavy defragmentation

Or in MYSQL detect fragmentation:

```
SELECT TABLE_SCHEMA, TABLE_NAME, CONCAT(ROUND(data_length / ( 1024 * 1024 ), 2), 'MB')
        DATA, CONCAT(ROUND(data_free  / ( 1024 * 1024 ), 2), 'MB')FREE
        from information_schema.TABLES where TABLE_SCHEMA NOT IN ('information_schema','mysql') and Data_free < 0;
```

TABLE_SCHEMA	TABLE_NAME	DATA	FREE
sys	x$wait_classes_global_by_latency	NULL	NULL
sys	x$waits_by_host_by_latency	NULL	NULL
sys	x$waits_by_user_by_latency	NULL	NULL
sys	x$waits_global_by_latency	NULL	NULL
test	covered	820.00MB	3.00MB
test	coveringidxexample	6.52MB	4.00MB
test	dateex	86.63MB	0.00MB
test	delimiter	0.02MB	0.00MB
test	minimal_select	0.47MB	0.00MB
test	predicateexample	48.58MB	13.00MB

Maximizing data in pages

- Index records inserted randomly will be between MERGE FACTOR % full and FILL FACTOR % full.
- Sorted indexes can be filled to their fill factor
- Use a sorted index build (mysql) or an index rebuild (sql server) to eliminate fragmentation
  - Compress data pages -> fewer pages reads
  - Use “ONLINE” option to continue OLTP

Bulk Load

Bulk load prevents page splits because it can put the data in sorted order and fix the index after the load. If you don't use bulk load, it has to maintain the index after every insertion which can lead to page splits.

- Same applies for single statements that specify multi-row inserts

Bulk load doesn't have the overhead in keeping maintaining point in time restore.

Ex: MySQL: LOAD DATA INFILE 'C:\datafile'

SQL Server: BULK INSERT dbname.schemaname.tablename FROM 'C:\datafile'

Table Partitioning

Table Partitioning

Partitioning allows you to split your data into multiple sets to place on different disks in order to utilize all disk heads. This gives you parallel performance.

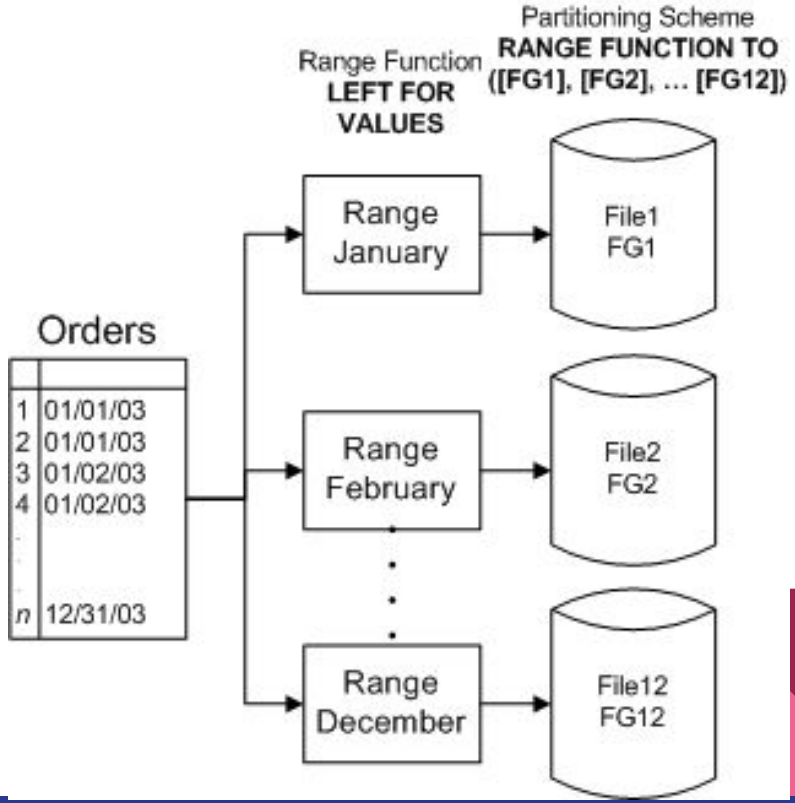


Table Partitioning

Types
1. Range or value lists
2. Hash function

Example: One Partition per month:

```
CREATE TABLE ti (id INT, amount DECIMAL(7,2), tr_date DATE)
    ENGINE=INNODB
    PARTITION BY HASH( MONTH(tr_date) )
    PARTITIONS 12;
```

```
SQL Server: Or 4 partitions (-Inf, 1], (1, 100], (100, 1000],
(1000, Inf):
CREATE PARTITION FUNCTION myRangePF1 (int)
    AS RANGE LEFT FOR VALUES (1, 100, 1000) ;
GO
```