14.1 Physical design

MySQL storage engines

Logical design specifies tables, columns, and keys. The logical design process is described elsewhere in this material. ***Physical design*** specifies indexes, table structures, and partitions. Physical design affects query performance but never affects query results.

A ***storage engine*** or ***storage manager*** translates instructions generated by a query processor into low-level commands that access data on storage media. Storage engines support different index and table structures, so physical design is dependent on a specific storage engine.

MySQL can be configured with several different storage engines, including:

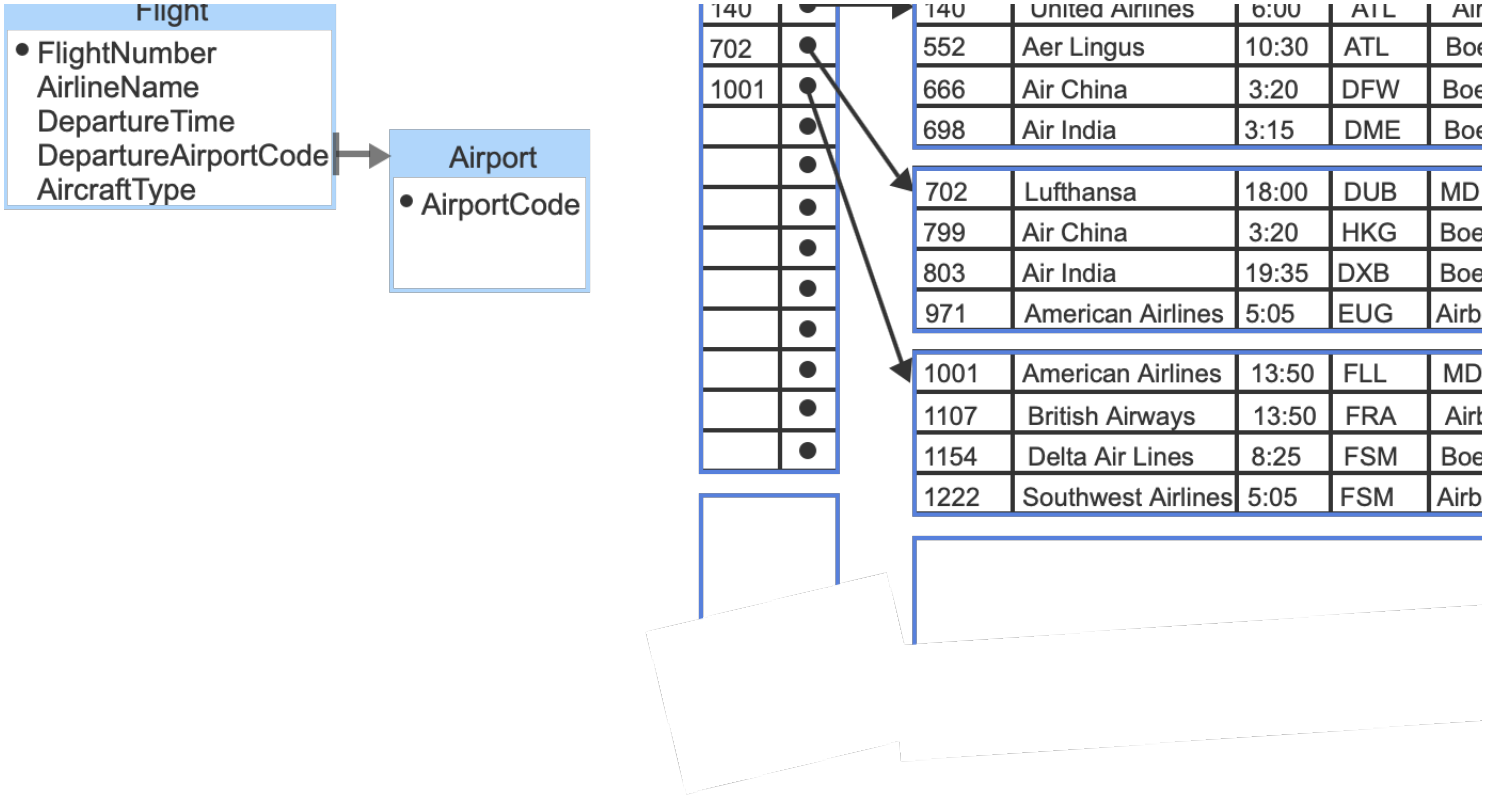
- *InnoDB* is the default storage engine installed with the MySQL download. InnoDB has full support for transaction management, foreign keys, referential integrity, and locking.
- *MyISAM* has limited transaction management and locking capabilities. MyISAM is commonly used for analytic applications with limited data updates.
- *MEMORY* stores all data in main memory. MEMORY is used for fast access with databases small enough to fit in main memory.

Different databases and storage engines support different table structures and index types. Ex:

- *Table structure*. Oracle Database supports heap, sorted, hash, and cluster tables. MySQL with InnoDB supports only heap and sorted tables.
- *Index type*. MySQL with InnoDB or MyISAM supports only B+tree indexes. MySQL with MEMORY supports both B+tree and hash indexes.

This section describes the physical design process and statements for MySQL with InnoDB. The process and statements can be adapted to other databases and storage engines, but details depend on supported index and table structures.

PARTICIPATION ACTIVITY 14.1.1: Logical and physical design.



Animation content:

Static figure:
A table diagram has caption logical design. A sorted table and index has caption physical design.

The table diagram has tables Flight and Airport. Flight has columns FlightNumber, AirlineName, DepartureTime, DepartureAirportCode, and AircraftType. FlightNumber is the primary key. Airport has column AirportCode. AirportCode is the primary key. An arrow points from DepartureAirportCode of the Flight table to the Airport table.

The sorted table has three blocks. Each block contains four rows. Each row contains flight number, airline name, departure time, airport code, and aircraft type. The rows are sorted on flight number.

The index has one level and is sparse. Index entries contain flight number and a pointer to the table block beginning with the corresponding row.

Step 1: Logical design specifies tables and columns. The logical design caption and Flight table appear.

Step 2: Logical design also specifies primary and foreign keys. The Airport table appears. Primary and foreign key columns are highlighted.

Step 3: Physical design specifies table structure. The Flight table is sorted on FlightNumber. The physical design caption and sorted table appear.

Step 4: Physical design also specifies indexes and index types. The index appears.

Animation captions:

1. Logical design specifies tables and columns.
2. Logical design also specifies primary and foreign keys.
3. Physical design specifies table structure. The Flight table is sorted on FlightNumber.
4. Physical design also specifies indexes and index types.

PARTICIPATION ACTIVITY 14.1.2: Logical and physical design.

Indicate whether each task is a logical or physical design activity.

1) Select a storage engine for MySQL.
   - ○ Logical design
   - ○ Physical design

2) Specify a column is UNIQUE.
   - ○ Logical design
   - ○ Physical design

3) Specify an index is CLUSTERED.
   - ○ Logical design
   - ○ Physical design

4) Determine foreign keys.
   - ○ Logical design
   - ○ Physical design

CREATE INDEX, DROP INDEX, and SHOW INDEX statements

In MySQL with InnoDB:

- Indexes are always B+tree indexes.
- A primary index is automatically created on every primary key.
- A secondary index is automatically created on every foreign key.
- Additional secondary indexes are created manually with the CREATE INDEX statement.
- Tables with a primary key have sorted structure. Tables with no primary key have a heap structure.

The ***CREATE INDEX*** statement creates an index by specifying the index name and table columns that compose the index. Most indexes specify just one column, but a composite index specifies multiple columns.

The ***DROP INDEX*** statement deletes a table's index.

The ***SHOW INDEX*** statement displays a table's index. SHOW INDEX generates a result table with one row for each column of each index. A multi-column index has multiple rows in the result table.

The SQL standard includes logical design statements such as CREATE TABLE but not physical design statements such as CREATE INDEX. Nevertheless, CREATE INDEX and many other physical design statements are similar in most relational databases.

Table 14.1.1: INDEX statements.

Statement	Description	Syntax
CREATE INDEX	Create an index	`CREATE INDEX IndexName ON TableName (Column1, Column2, ..., ColumnN);`
DROP INDEX	Delete an index	`DROP INDEX IndexName ON TableName;`
SHOW INDEX	Show an index	`SHOW INDEX FROM TableName;`

Table 14.1.2: SHOW INDEX result table (selected columns).

Column Name	Column Meaning
Table	Name of indexed table
	0 if index is on unique column

Non_unique	0 if index is on unique column 1 if index is on non-unique column
Key_name	Name of index as specified in CREATE INDEX statement or created by MySQL
Seq_in_index	1 for single-column indexes Numeric order of column in multi-column indexes
Column_name	Name of indexed column
Cardinality	Number of distinct values in indexed column
Null	YES if NULLs are allowed in column Blank if NULLs are not allowed in column
Index_type	Always BTREE for InnoDB storage engine

PARTICIPATION ACTIVITY 14.1.3: INDEX Statements.

Address

• AddressID	Street	District	CityID	PostalCode
1	1913 Hanoi Way	Nagasaki	463	35200
2	1121 Loja Avenue	California	449	17886
3	692 Joliet Street	Attika	38	83579
4	1566 Inegl Manor	Mandalay	349	53561
5	53 Idfu Parkway	Nantou	361	42399

```
CREATE INDEX PostalCodeIndex
ON Address (PostalCode);

SHOW INDEX FROM Address;
```

Table	Non_unique	Key_name	Seq_in_index	Column_name	Cardinality	Null	Index_type
Address	0	PRIMARY	1	AddressID	603		BTREE
Address	1	idx_fk_city_id	1	CityID	99		BTREE
Address	1	PostalCodeIndex	1	PostalCode	597	YES	BTREE

(selected columns)

Animation content:

Static figure:
The Address table has columns AddressID, Street, District, CityID, and PostalCode. AddressID is the primary key. Address has five rows.

Two SQL statements appear.
Begin SQL code:
CREATE INDEX PostalCodeIndex
ON Address (PostalCode);

SHOW INDEX FROM Address;
End SQL code.

The result table for the SHOW statement appears, with column Table, Non_unique, Key_name, Seq_in_index, Column_name, Cardinality, Null, and Index_type. The table has caption (selected columns). The table has three rows. The first two rows have no value in column Null:
Address, 0, PRIMARY, 1, AddressID, 603, , BTREE
Address, 0, idx_fk_city_id, 1, CityID, 99, , BTREE
Address, 0, PostalCodeIndex, 1, PostalCode, 597, YES, BTREE

Step 1: The Address table has primary key AddressID and foreign key CityID. The Address table appears.

Step 2: The CREATE INDEX statement creates a secondary index on PostalCode. The PostalCode column is highlighted. The CREATE INDEX statement appears.

Step 3: The SHOW INDEX statement displays all indexes on the Address table. Some columns of the result table have been omitted. The SHOW INDEX statement and result table appear.

Step 4: MySQL automatically creates indexes on primary and foreign keys. AddressID has 603 distinct values. CityID has 99 distinct values. The first two result rows are highlighted.

Step 5: PostalCodeIndex was created with the CREATE INDEX statement. PostalCode is not unique and has 597 distinct values. The third result row is highlighted.

Animation captions:

1. The Address table has primary key AddressID and foreign key CityID.
2. The CREATE INDEX statement creates a secondary index on PostalCode.
3. The SHOW INDEX statement displays all indexes on the Address table. Some columns of the result table have been omitted.

the result table have been omitted.

4. MySQL automatically creates indexes on primary and foreign keys. AddressID has 603 distinct values. CityID has 99 distinct values.
5. PostalCodeIndex was created with the CREATE INDEX statement. PostalCode is not unique and has 597 distinct values.

PARTICIPATION ACTIVITY | 14.1.4: INDEX statements.

Indexes are created on the Flight table as follows:

```
CREATE INDEX FirstIndex
ON Flight (FlightNumber, AirlineName);

CREATE INDEX SecondIndex
ON Flight (DepartureAirportCode);
```

The questions below refer to the result table of the following statement:

```
SHOW INDEX FROM Flight;
```

1) How many result table rows have 'FirstIndex' in the Key_name column?

Check Show answer

2) FlightNumber is the primary key of Flight. Flight has 90 rows. In the row with Column_name = "FlightNumber", what is the value of Cardinality?

Check Show answer

3) In the row with Column_name = "AirlineName", what is the value of Seq_in_index?

Check Show answer

EXPLAIN statement

The ***EXPLAIN*** statement generates a result table that describes how a statement is executed by the storage engine. EXPLAIN syntax is simple and uniform in most databases: `EXPLAIN statement;` The `statement` can be any SELECT, INSERT, UPDATE, or DELETE statement.

Although the EXPLAIN statement is supported by most relational databases, the result table varies significantly. In MySQL with InnoDB, the result table has one row for each table in the statement. If the statement contains multiple queries, such as a main query and a subquery, the result table has one row for each table in each query.

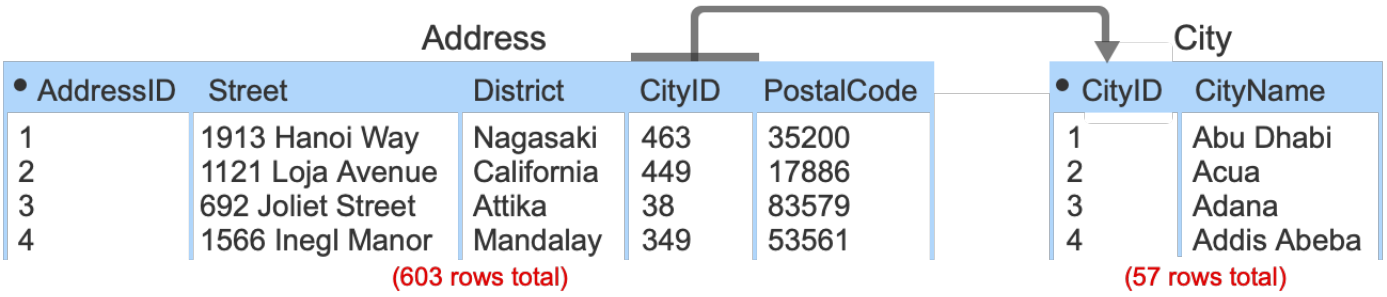
The type column of the EXPLAIN result table indicates how MySQL processes a join query. Processing join queries efficiently is a complex problem, so the type column has many alternative values. Example values appear in the table below. For more information, see MySQL EXPLAIN result table.

Table 14.1.3: EXPLAIN result table (selected columns).

Column Name	Column Meaning
select_type	The query type. Example query types: *SIMPLE* indicates query is neither nested nor union *PRIMARY* indicates query is the outer SELECT of nested query *SUBQUERY* indicates query is an inner SELECT of nested query
table	Name of table described in row of EXPLAIN result table
type	The join type. Example join types: *const* indicates the table has at most one matching row *range* indicates a join column is compared to a constant using operators such as BETWEEN, LIKE, or IN() *eq_ref* indicates one table row is read for each combination of rows from other tables (typically, an equijoin) *ALL* indicates a table scan is executed for each combination of rows from other tables
possible_keys	All available indexes that might be used to process the query
	The index selected to process the query

key	The index selected to process the query *NULL* indicates a table scan is performed
ref	The constant, column, or expression to which the selected index is compared
rows	Estimated number of rows read from table
filtered	Estimated number of rows selected by WHERE clause / estimated number of rows read from table

PARTICIPATION ACTIVITY 14.1.5: EXPLAIN statement.




```
EXPLAIN SELECT Street, CityName, PostalCode
        FROM Address, City
        WHERE Address.CityID = City.CityID AND Address.PostalCode > 40000;
```

Join result

Street	CityName	PostalCode
53 Idfu Parkway	Nantou	42399
613 Korolev Drive	Masqat	45844
419 Iligan Lane	Bhopal	72878
320 Brest Avenue	Kaduna	43331
96 Tafuna Way	Crdoba	99865

Explain result

select_type	table	type	possible_keys	key	ref	rows	filte
SIMPLE	Address	ALL	idx_fk_city_id, PostalCodeIndex	NULL	NULL	603	33.
SIMPLE	City	eq_ref	PRIMARY	PRIMARY	sakila.address.city_id	1	100

(selected columns)

Animation content:

Static figure:
The Address table has columns AddressID, Street, District, CityID, and PostalCode. AddressID is the primary key. Address has four rows and caption (603 rows total). The City table has columns CityID and CityName. CityID is the primary key. City has four rows and caption (57 rows total). An arrow points from column CityID of the Address table to column CityID of the City table.

An SQL statement appears.
Begin SQL code:

```
EXPLAIN SELECT Street, CityName, PostalCode
    FROM Address, City
    WHERE Address.CityID = City.CityID AND Address.PostalCode > 40000;
```

End SQL code.

A Join result table has columns Street, CityName, and PostalCode. Join result is the result of the SELECT statement embedded in the EXPLAIN statement. Join result has five rows.

An Explain result table, with caption (selected columns), has columns select_type, table, type, possible_keys, key, ref, rows, and filtered. Explain result has two rows:
SIMPLE, Address, ALL, idx_fk_city_id PostalCodeIndex, NULL, NULL, 603, 33.3
SIMPLE, City, eq_ref, PRIMARY, PRIMARY, sakila.address.city_id, 1, 100.00

Step 1: Address table has 603 rows. City table has 57 rows. The Address and City tables appear with captions and arrow.

Step 2: The SELECT statement joins Address and City tables. The SELECT statement appears without the EXPLAIN keyword. The Join result table appears.

Step 3: EXPLAIN generates a results table with one row for each table in the SELECT statement. The EXPLAIN keyword appears prior to the SELECT statement. The Explain result table appears. Values in the column table are highlighted.

Step 4: Since 33% of Address rows are selected, hit ratio is high. A table scan (type = ALL) is more efficient than an index scan. In the first row of Explain result, values in columns table, type, key, and filtered are highlighted:
Address, ALL, NULL, 33.33

Step 5: The City table is accessed via primary index. One City row is read and selected for each Address row. In the second row of Explain result, values in columns table, key, rows, and filtered are highlighted:
City, PRIMARY, 1, 100.00

City, PRIMARY, 1, 100.00

Animation captions:

1. Address table has 603 rows. City table has 57 rows.
2. The SELECT statement joins Address and City tables.
3. EXPLAIN generates a results table with one row for each table in the SELECT statement.
4. Since 33% of Address rows are selected, hit ratio is high. A table scan (type = ALL) is more efficient than an index scan.
5. The City table is accessed via primary index. One City row is read and selected for each Address row.

PARTICIPATION ACTIVITY | 14.1.6: EXPLAIN statement.

The Flight table has 90 rows and primary key FlightNumber. No CREATE INDEX statements are executed on Flight. The questions below refer to the result table of the following statement:

```
EXPLAIN
  SELECT FlightNumber
  FROM Flight
  WHERE AirportCode IN
      (SELECT AirportCode
      FROM Airport
      WHERE Country = "Cuba");
```

1) In the row with table = "Airport", what is the value of select_type?

Check Show answer

2) In the row with table = "Flight", the _____ column contains NULL because no indexes exist that might be used to process the query.

Check Show answer

3) Assume the Airport table contains an AirportCode for Cuba. In the row with table = "Flight", what is the value of rows?

Check Show answer

Physical design process

A database administrator may take a simple approach to physical design for MySQL with InnoDB:

1. *Create initial physical design*. Create a primary index on primary keys and a secondary index on foreign keys. In MySQL with InnoDB, these indexes are created automatically for all tables. In other databases, this step is necessary for tables larger than roughly 100 kilobytes, but can be omitted for smaller tables.
2. *Identify slow queries*. The MySQL ***slow query log*** is a file that records all long-running queries submitted to the database. Identify slow queries by inspecting the log. Most other relational databases have similar query logs.
3. *EXPLAIN slow queries*. Run EXPLAIN on each slow query to assess the effectiveness of indexes. A high value for *rows* and a low value for *filtered* indicates either a table scan or an ineffective index.
4. *Create and drop indexes* based on the EXPLAIN result table. Consider creating an index when the *rows* value is high and the *filtered* value is low. Consider dropping indexes that are never used.
5. *Partition large tables*. If some queries are still slow after indexes are created, consider partitions. Partition when slow queries access a small subset of rows of a large table. The partition column should appear in the WHERE clause of slow queries. Often, a range partition is best.

Steps 2 through 5 are ongoing activities. As the database grows and usage increases, the database administrator periodically reviews the query log, runs EXPLAIN, and adjusts the physical design for optimal performance. Additional tuning techniques can be found in the "Exploring further" section.

The five steps above can be adapted to other databases and storage engines, but the details depend on supported table structures, index types, and partition types.

PARTICIPATION | 14.1.7: Physical design for MySQL with InnoDB

Address

• AddressID	Street	District	CityID	PostalCode
1	1913 Hanoi Way	Nagasaki	463	35200
2	1121 Loja Avenue	California	449	17886
3	692 Joliet Street	Attika	38	83579
4	1566 Inegl Manor	Mandalay	349	53561

(603 rows total)

City

• CityID	CityName
1	Abu Dhabi
2	Acua
3	Adana
4	Addis Abeba

(600 rows total)

slow query log

```
EXPLAIN SELECT District
        FROM Address, City
        WHERE Address.CityID = City.CityID AND CityName = "Aurora";
```

select_type	table	type	possible_keys	key	ref	rows	filtered
SIMPLE	City	ALL	PRIMARY	NULL	NULL	600	10.00
SIMPLE	Address	ref	idx_fx_city_id	idx_fk_city_id	sakila.city.city_id	1	100.00

(selected columns)

```
CREATE INDEX CityNameIndex
ON City (CityName);
```

select_type	table	type	possible_keys	key	ref	rows	filtered
SIMPLE	City	ref	PRIMARY, CityNameIndex	CityNameIndex	const	1	100.00
SIMPLE	Address	ref	idx_fx_city_id	idx_fk_city_id	sakila.city.city_id	1	100.00

(selected columns)

Animation content:

Static figure:
The Address table has columns AddressID, Street, District, CityID, and PostalCode. AddressID is the primary key. Address has four rows and caption (603 rows total). The City table has columns CityID and CityName. CityID is the primary key. City has four rows and caption (600 rows total). An arrow points from column CityID of the Address table to column CityID of the City table.

An SQL statement appears with caption slow query log.
Begin SQL code:
EXPLAIN SELECT District
FROM Address, City

FROM Address, City
WHERE Address.CityID = City.CityID AND CityName = 'Aurora';
End SQL code.

A result table, with caption (selected columns), has columns select_type, table, type, possible_keys, key, ref, rows, and filtered. The result table has two rows:
SIMPLE, City, ALL, PRIMARY, PRIMARY, NULL, 600, 10.00
SIMPLE, Address, ref, idx_fk_city_id, idx_fk_city_id, sakila.city.city_id, 1, 100.00
In the first row, values in columns key, rows, and filtered are highlighted:
NULL, 600, 10.00

A second SQL statement creates an index.
Begin SQL code:
CREATE INDEX CityNameIndex
ON City (CityName);
End SQL code.

A second result table, with caption (selected columns), has columns select_type, table, type, possible_keys, key, ref, rows, and filtered. The result table has two rows:
SIMPLE, City, ALL, PRIMARY CityNameIndex, CityNameIndex, const, 1, 100.00
SIMPLE, Address, ref, idx_fk_city_id, idx_fk_city_id, sakila.city.city_id, 1, 100.00
In the first row, values in columns key, rows, and filtered are highlighted:
CityNameIndex, 1, 100.00

Step 1: The SELECT statement appears in the MySQL slow query log. The Address and City tables appear, with captions and arrow. The first SQL statement appears, with caption but without the EXPLAIN keyword.

Step 2: The database administrator runs EXPLAIN on the SELECT query. The first result table appears.

Step 3: The EXPLAIN result shows no index was used, many rows were read, and few rows were selected. In the first row of the result, values in columns key, rows, and filtered are highlighted:
NULL, 600, 10.00

Step 4: To speed up the SELECT query, the database administrator creates an index on the CityName column. The CREATE INDEX statement appears.

Step 5: Running EXPLAIN again shows the query uses CityNameIndex, reads only one row, and executes faster. The second result table appears. In the first row, values in columns key, rows, and filtered are highlighted:
CityNameIndex, 1, 100.00

Animation captions:

1. The SELECT statement appears in the MySQL slow query log.
2. The database administrator runs EXPLAIN on the SELECT query.
3. The EXPLAIN result shows no index was used, many rows were read, and few rows were selected.
4. To speed up the SELECT query, the database administrator creates an index on the CityName column.
5. Running EXPLAIN again shows the query uses CityNameIndex, reads only one row, and executes faster.

PARTICIPATION ACTIVITY 14.1.8: Physical design process for MySQL.

1) An index never appears in the key column of the EXPLAIN result table for any slow query. Should this index always be dropped?

   - ○ Yes
   - ○ No

2) In the EXPLAIN result table for one slow query, the key column is NULL, the filtered column is 1%, and the rows column is 10 million. Should an index be created?

   - ○ Yes
   - ○ No

3) Additional indexes never slow performance of SELECT queries.

   - ○ True
   - ○ False

4) In step 1 of physical design for MySQL with InnoDB, the database administrator determines table structures.

   - ○ True

- False

Exploring further:

- MySQL InnoDB storage engine
- MySQL CREATE INDEX syntax
- MySQL EXPLAIN result table
- MySQL slow query log
- Optimizing MySQL queries with EXPLAIN

14.2 Single-level indexes

Single-level indexes

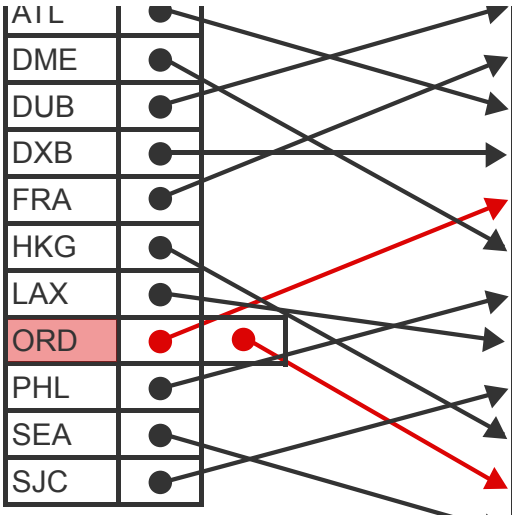
A ***single-level index*** is a file containing column values, along with pointers to rows containing the column value. The pointer identifies the block containing the row. In some indexes, the pointer also identifies the exact location of the row within the block. Indexes are created by database designers with the CREATE INDEX command, described elsewhere in this material.

Single-level indexes are normally sorted on the column value. A sorted index is not the same as an index on a sorted table. Ex: An index on a heap table is a sorted index on an unsorted table.

If an indexed column is unique, the index has one entry for each column value. If an indexed column is not unique, the index may have multiple entries for some column values, or one entry for each column value, followed by multiple pointers.

An index is usually defined on a single column, but an index can be defined on multiple columns. In a ***multi-column index***, each index entry is a composite of values from all indexed columns. In all other respects, multi-column indexes behave exactly like indexes on a single column.

PARTICIPATION ACTIVITY 14.2.1: Single-level index.



ATL		140	Aer Lingus	10:30	DUB	Boeing 737
DME		552	Lufthansa	18:00	FRA	MD 111
DUB		666	United Airlines	6:00	ATL	Airbus 321
DXB		698	Air India	3:15	DXB	Boeing 747
FRA		702	American Airlines	5:05	ORD	Airbus 323
HKG		799	Air China	3:20	DME	Boeing 777
LAX		803	British Airways	13:50	PHL	Airbus 323
ORD		971	Air India	19:35	LAX	Boeing 747
PHL		1001	Southwest Airlines	5:05	SJC	Airbus 323
SEA		1107	Air China	3:20	HKG	Boeing 787
SJC		1154	American Airlines	13:50	ORD	MD 111
		1222	Delta Airlines	8:25	SEA	Boeing 747

Animation content:

Step 1: An index is on the DepartureAirport column of the Flight table. A 5-column flight index table has the column of destination city abbreviations highlighted.

Step 2: Each column value appears in the index in sorted order. The city abbreviations are sorted into an alphabetic single-level index. The value ORD in the single-level index appears twice .

Step 3: Each column value has a pointer to the row containing the value. The single-level index values connect to their matching rows in the flight index table via pointers.

Step 4: If the column values are not unique, the index may have multiple entries for some values. The ORD values in the single-level index each connect to a different row in the table.

Step 5: Alternatively, an index on a non-unique column may have multiple pointers for some values. The second entry of ORD in the single-level index is eliminated, leaving one ORD value that now has two pointers to the flight index table.

Animation captions:

1. An index is on the DepartureAirport column of the Flight table.
2. Each column value appears in the index in sorted order.
3. Each column value has a pointer to the row containing the value.
4. If the column values are not unique, the index may have multiple entries for some values.
5. Alternatively, an index on a non-unique column may have multiple pointers for some values.

PARTICIPATION ACTIVITY | 14.2.2: Single-level indexes.

Refer to the index and table in the animation above.

1) Which index entry refers to Air China flight 1107?

- The first entry
- The sixth entry
- The eleventh entry

2) Lufthansa flight 44, departing from Munich, is inserted into the table. The code for Munich is MUC. Where does the new index entry go?

- At the end of the index.
- At the first available free space in the index.
- Between the LAX and ORD index entries.

3) An indexed column requires 12 bytes. Pointers to table blocks require 8 bytes. Index blocks are 4 kilobytes. Approximately how many rows can be referenced in one index block?

- 200
- 400
- Unlimited

4) An index on a non-unique column can either:
A) store multiple pointers after one value, or
B) store duplicate values with one pointer each.
What is the advantage of strategy A?

- Index is more compact.
- Index has variable length entries.

Query processing

To execute a SELECT query, the database can perform a table scan or an index scan:

- A ***table scan*** is a database operation that reads table blocks directly, without accessing an index.
- An ***index scan*** is a database operation that reads index blocks sequentially, in order to locate the needed table blocks.

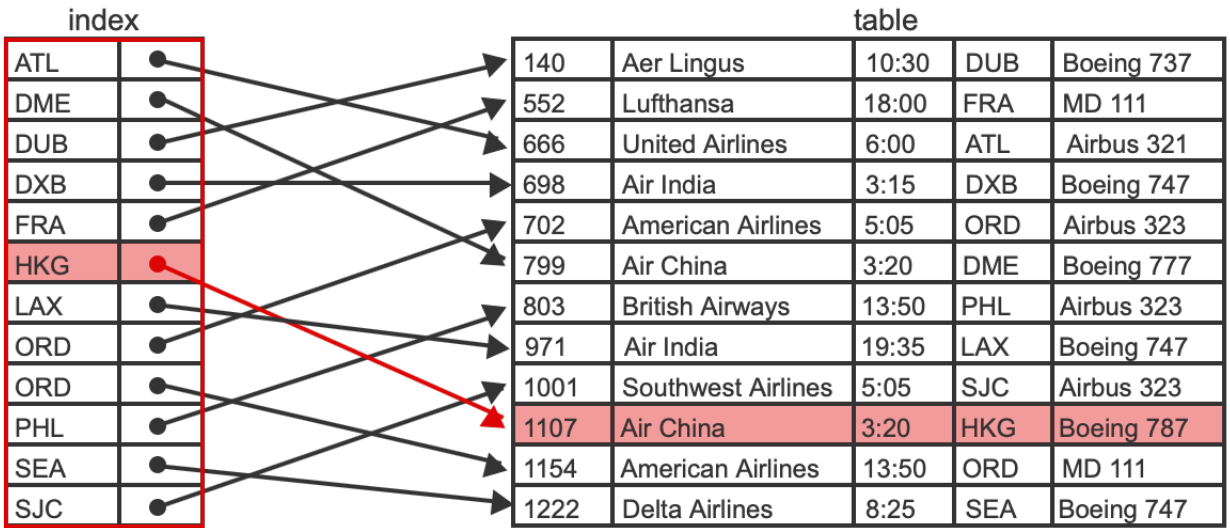
Hit ratio, also called ***filter factor*** or ***selectivity***, is the percentage of table rows selected by a query. When a SELECT query is executed, the database examines the WHERE clause and estimates hit ratio. If hit ratio is high, the database performs a table scan. If hit ratio is low, the query needs only a few table blocks, so a table scan would be inefficient. Instead, the database:

1. Looks for an indexed column in the WHERE clause.
2. Scans the index.
3. Finds values that match the WHERE clause.
4. Reads the corresponding table blocks.

If the WHERE clause does not contain an indexed column, the database must perform a table scan.

Since a column value and pointer occupy less space than an entire row, an index requires fewer blocks than a table. Consequently, index scans are much faster than table scans. In some cases, indexes are small enough to reside in main memory, and index scan time is insignificant. When hit ratio is low, additional time to read the table blocks containing selected rows is insignificant.

PARTICIPATION ACTIVITY 14.2.3: Query processing with single-level index.



```
SELECT FlightNumber, AirlineName
FROM Flight;
```

FlightNumber	AirlineName
140	Aer Lingus
552	Lufthansa
666	United Airlines
698	Air India
702	American Airlines
799	Air China
803	British Airways
971	Air India
1001	Southwest Airlines
1107	Air China
1154	American Airlines
1222	Delta Airlines

```
SELECT FlightNumber, AirlineName
FROM Flight
WHERE FlightNumber = 803;
```

FlightNumber	AirlineName
803	British Airways

```
SELECT FlightNumber, AirlineName
FROM Flight
WHERE DepartureAirport = 'HKG';
```

FlightNumber	AirlineName
1107	Air China

Animation content:

Static figure:
An index and table appear. The index has entries containing airport code and a pointer to a table row. Index entries are sorted by airport code. The table has twelve rows containing flight number, airline name, airport code, and other data. The table rows are sorted by flight number and represent data storage.

Three statements appear. Each query is followed by a result table. All result tables have columns FlightNumber and AirlineName.

The first statement is:
Begin SQL code:
SELECT FlightNumber, AirlineName
FROM Flight;
End SQL code.

The first result includes all rows of the table.

The second statement is:
Begin SQL code:
SELECT FlightNumber, AirlineName
FROM Flight
WHERE FlightNumber = 803;
End SQL code.

The second result has one row:

The second result has one row:
803, British Airways

The third statement is:
Begin SQL code:
SELECT FlightNumber, AirlineName
FROM Flight
WHERE DepartureAirport = 'HKG';
End SQL code.

The third result has one row:
1107, AirChina

Step 1: FlightNumber is the Flight table's primary key. FlightNumber is sorted but is not indexed. The index and table appear. The flight number data in the table is highlighted.

Step 2: The SELECT statement has no WHERE clause, so the hit ratio is high. The database performs a table scan and reads all table blocks. The first statement appears. All three blocks of table data, labeled 0, 1 and 2, are highlighted. The first result appears.

Step 3: The WHERE clause does not include an indexed column, so the database performs a table scan. The second statement appears. The WHERE clause is highlighted.

Step 4: Only the first two blocks are read because FlightNumber 803 is found in the second table block. Blocks 0 and 1 of table data are highlighted. The row containing flight number 803 is in block 1 and is highlighted. The second result appears.

Step 5: The WHERE clause contains an indexed column. Since the hit ratio is low, the database performs an index scan. The third query appears. Index entry HKG is highlighted. This index entry points to the table row containing HKG, which is highlighted. The third result appears.

Animation captions:

1. FlightNumber is the Flight table's primary key. FlightNumber is sorted but is not indexed.
2. The SELECT statement has no WHERE clause, so the hit ratio is high. The database performs a table scan and reads all table blocks.
3. The WHERE clause does not include an indexed column, so the database performs a table scan.
4. Only the first two blocks are read because FlightNumber 803 is found in the second table block.
5. The WHERE clause contains an indexed column. Since the hit ratio is low, the database performs an index scan.

PARTICIPATION ACTIVITY 14.2.4: Query processing with single-level indexes.

Refer to the following scenario:

- A table occupies 2,000 blocks.
- FlightNumber is the primary key.
- An index on FlightNumber occupies 200 blocks.
- The WHERE clause of a SELECT specifies "FlightNumber = 3988".

The query is executed with an index scan.

1) What is the maximum number of blocks necessary to process the SELECT?
   - ○ 2
   - ○ 201
   - ○ 2,000

2) What is the minimum number of blocks necessary to process the SELECT?
   - ○ 1
   - ○ 2
   - ○ 201

3) Assume the table has no index. What is the maximum number of blocks necessary to process the SELECT?
   - ○ 1
   - ○ 201
   - ○ 2,000

Binary search

When hit ratio is low, index scans are always faster than table scans. Consider the following scenario:

- A table has 10 million rows.
- Each row is 100 bytes.
- Each block is 4 kilobytes.
- Each index entry is 10 bytes, including a 6-byte value and a 4-byte pointer.
- Magnetic disk transfer rate is 0.1 gigabytes per second.

If hit ratio is low, an index scan is roughly 10 times faster than a table scan:

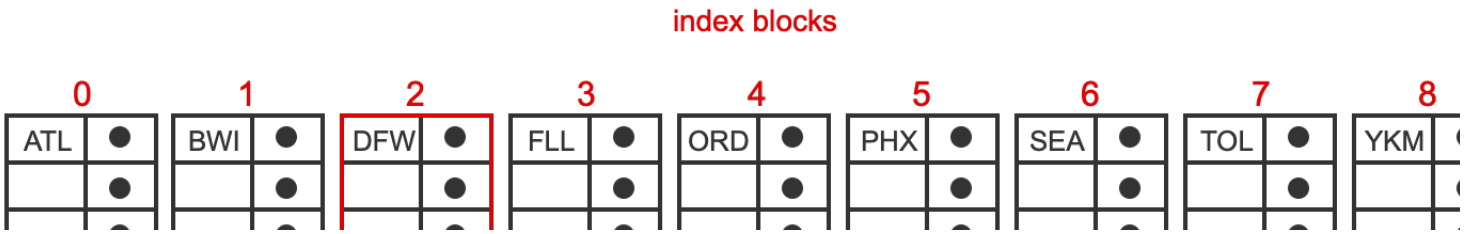
- *Table scan*. The table contains 1 billion bytes (10 million rows × 100 bytes/row). The table scan takes around 10 seconds (1 gigabyte / 0.1 gigabytes/sec).
- *Index scan*. The index contains 100 million bytes (10 million rows × 1 entry/row × 10 bytes/entry). The index scan takes around 1 second (0.1 gigabyte / 0.1 gigabytes/sec). The index scan returns pointers to blocks containing rows selected by the query. When hit ratio is low, additional time to read table blocks is insignificant.

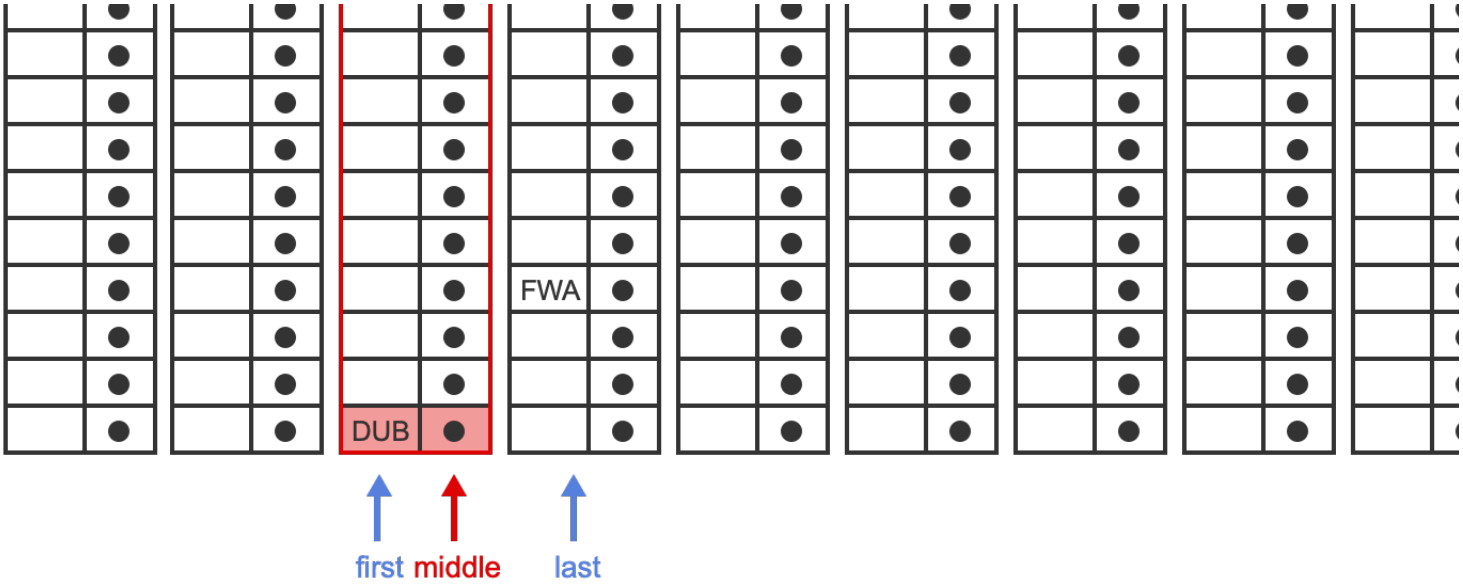
Although index scans are faster than table scans, index scans are too slow in many cases. If a single-level index is sorted, each value can be located with a binary search. In a ***binary search***, the database repeatedly splits the index in two until it finds the entry containing the search value:

1. The database first compares the search value to an entry in the middle of the index.
2. If the search value is less than the entry value, the search value is in the first half of the index. If not, the search value is in the second half.
3. The database now compares the search value to the entry in the middle of the selected half, to narrow the search to one quarter of the index.
4. The database continues in this manner until it finds the index block containing the search value.

For an index with N blocks, a binary search reads $\log_2$ N blocks, rounded up to the nearest integer. In the example above, the index has 25,000 blocks (10,000,000 rows × 10 bytes/index entry / 4,000 bytes/ block) . The binary search reads at most $\log_2$ 25,000, rounded up, or 15 blocks. This search takes about 0.0006 seconds (15 blocks × 4 kilobytes/block / 0.1 gigabytes/sec).

PARTICIPATION ACTIVITY 14.2.5: Binary search on a sorted index.






```
SELECT FlightNumber, AirlineName
FROM Flight
WHERE DepartureAirport = 'DUB';
```

Animation content:

Static figure:
Nine index blocks appear, labeled 0 through 8. Index entries contain an airport code and pointer to a table row. The first entry of blocks 0 through 8 contains, respectively, airport code ATL, BWI, DFW, FLL, ORD, PHX, SEA, TOL, and YKM. The last entry of block 2 contains DUB. A middle entry of block 3 contains FWA.

An SQL statement appears.
Begin SQL code:
SELECT FlightNumber, AirlineName
FROM Flight
WHERE DepartureAirport = 'DUB';
End SQL code.

Step 1: The DepartureAirport index occupies nine blocks. The SELECT query looks for the row containing DepartureAirport = 'DUB'. The index blocks and SQL statement appears. The WHERE clause is highlighted.

Step 2: A binary search first examines the middle, block 4, of the sorted index. Block 0 is labeled first. Block 4 is labeled middle. Block 8 is labeled last.

Step 3: Since 'DUB' precedes 'ORD', the binary search resets last to block 3 and middle to block 1. Block 0 is labeled first. Block 1 is labeled middle. Block 3 is labeled last.

Step 4: Since 'DUB' follows 'BWI', search resets first and middle to block 2. Block 2 labeled first and middle. Block 3 is labeled last.

Step 5: The binary search reads block 2 and finds 'DUB'. The database follows DUB's pointer to the table block containing the row. The last entry in block 2, containing DUB, is highlighted.

Animation captions:

1. The DepartureAirport index occupies nine blocks. The SELECT query looks for the row containing DepartureAirport = 'DUB'.
2. A binary search first examines the middle, block 4, of the sorted index.
3. Since 'DUB' precedes 'ORD', the binary search resets last to block 3 and middle to block 1.
4. Since 'DUB' follows 'BWI', search resets first and middle to block 2.
5. The binary search reads block 2 and finds 'DUB'. The database follows DUB's pointer to the table block containing the row.

PARTICIPATION ACTIVITY 14.2.6: Binary search.

Refer to the following scenario:

- A table has 100 million rows.
- Each row is 400 bytes.
- Each block is 8 kilobytes.
- Each index entry is 20 bytes.
- Magnetic disk transfer rate is 0.1 gigabytes per second.

1) Assuming no free space, a table scan requires approximately how many seconds?

   - 0.4
   - 40
   - 400

2) Assuming the index is sorted, a binary search for one row reads approximately how many blocks?

- ○ $\log_2$ 250,000
- ○ $\log_2$ 5,000,000
- ○ $\log_{10}$ 250,000

Primary and secondary indexes

Indexes on a sorted table may be primary or secondary:

- A ***primary index***, also called a ***clustering index***, is an index on a sort column.
- A ***secondary index***, also called a ***nonclustering index***, is an index that is not on the sort column.

A sorted table can have only one sort column, and therefore only one primary index. Usually, the primary index is on the primary key column(s). In some database systems, the primary index may be created on any column. Tables can have many secondary indexes. All indexes of a heap or hash table are secondary, since heap and hash tables have no sort column.

Indexes may also be dense or sparse:

- A ***dense index*** contains an entry for every table row.
- A ***sparse index*** contains an entry for every table block.

When a table is sorted on an index column, the index may be sparse, as illustrated in the animation below. Primary indexes are on sort columns and usually sparse. Secondary indexes are on non-sort columns and therefore are always dense.

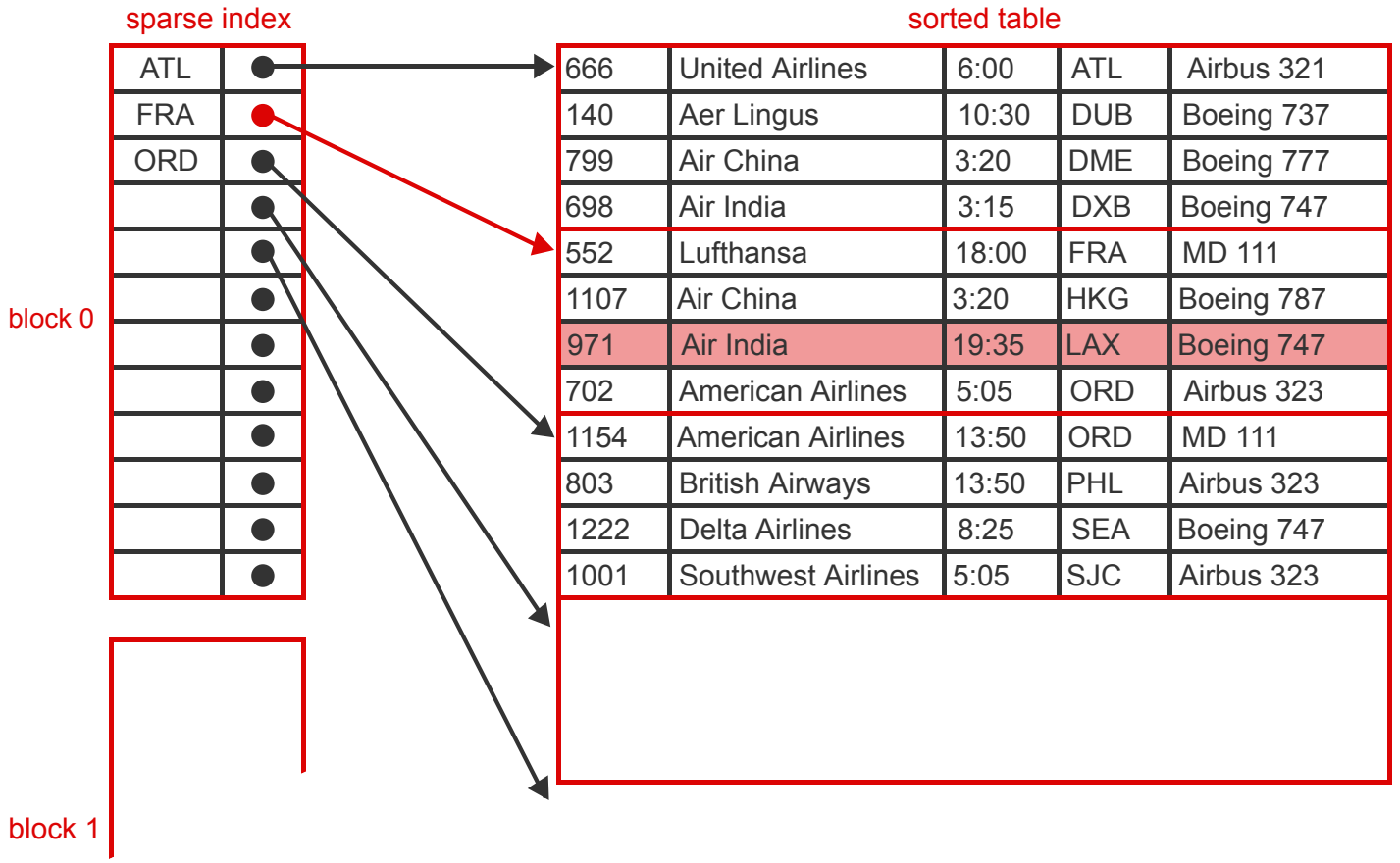
Sparse indexes are much faster than dense indexes since sparse indexes have fewer entries and occupy fewer blocks. Consider the following scenario:

- A table has 10 million rows.
- Each row is 100 bytes.
- Table and index blocks are 4 kilobytes.
- Each index entry is 10 bytes.

The table occupies 250,000 blocks (10 million rows × 100 bytes/row / 4 kilobytes/block). A sparse index requires 250,000 entries (one entry per table block) and occupies 625 blocks (250,000 entries × 10 bytes/entry / 4 kilobytes/block). The sparse index can easily be retained in main memory.

Primary indexes are usually sparse and sparse indexes are fast. As a result, database designers usually create a primary index on the primary key of large tables.

PARTICIPATION ACTIVITY | 14.2.7: Dense and sparse indexes.



Animation content:

Step 1: The table is not sorted on DepartureAirport column. The index on DepartureAirport must be dense. A heap table has four blocks, labeled 0 through 3. Block 3 is empty. A dense index has two blocks, labeled 0 and 1. The dense index has one entry for each row of the heap table.

Step 2: The table is sorted on DepartureAirport column. The index disappears. Rows of the heap table are sorted by the fourth column, airport code. The label heap table becomes sorted table.

Step 3: The index on the sort column may be sparse. The index appears with caption sparse index. The sparse index has three entries. Each entry points to the first row of a block in the sorted table.

Step 4: 'LAX' does not appear in the index, but the block containing 'LAX' can be determined from sort order. The airport code LAX appears between index entries for FRA and ORD. Index entry FRA points to block 1, which begins with a row containing FRA. Block 1 is highlighted. The row

containing LAX is in block 1 and is highlighted.

Animation captions:

1. The table is not sorted on DepartureAirport column. The index on DepartureAirport must be dense.
2. The table is sorted on DepartureAirport column.
3. The index on the sort column may be sparse.
4. 'LAX' does not appear in the index, but the block containing 'LAX' can be determined from sort order.

PARTICIPATION ACTIVITY | 14.2.8: Primary and secondary indexes.

Match the term with the term's description.

If unable to drag and drop, refresh the page.

Sparse index **Primary index** **Dense index** **Secondary index**

	Index with one entry for each table row.
	Index with one entry for each table block.
	An index on the table sort column.
	An index that is not on the table sort column.

Reset

Terminology

*The meanings of **primary index** and **clustering index** vary. In some database systems, primary and clustering indexes are indexes on unique and non-unique sort columns, respectively. In this material, the terms are synonymous and refer to an*

index on any sort column.

*A **clustering index** is not the same as a **cluster key**. A cluster key refers to a table cluster storage structure, described elsewhere in this material, and is not an index.*

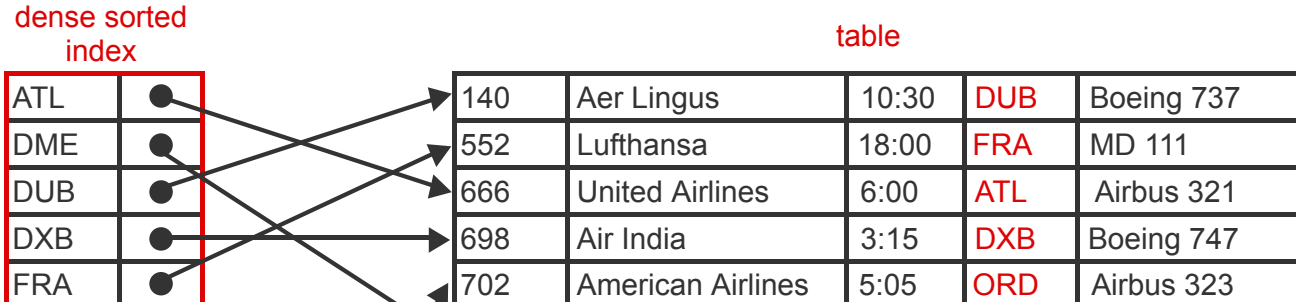
Inserts, updates, and deletes

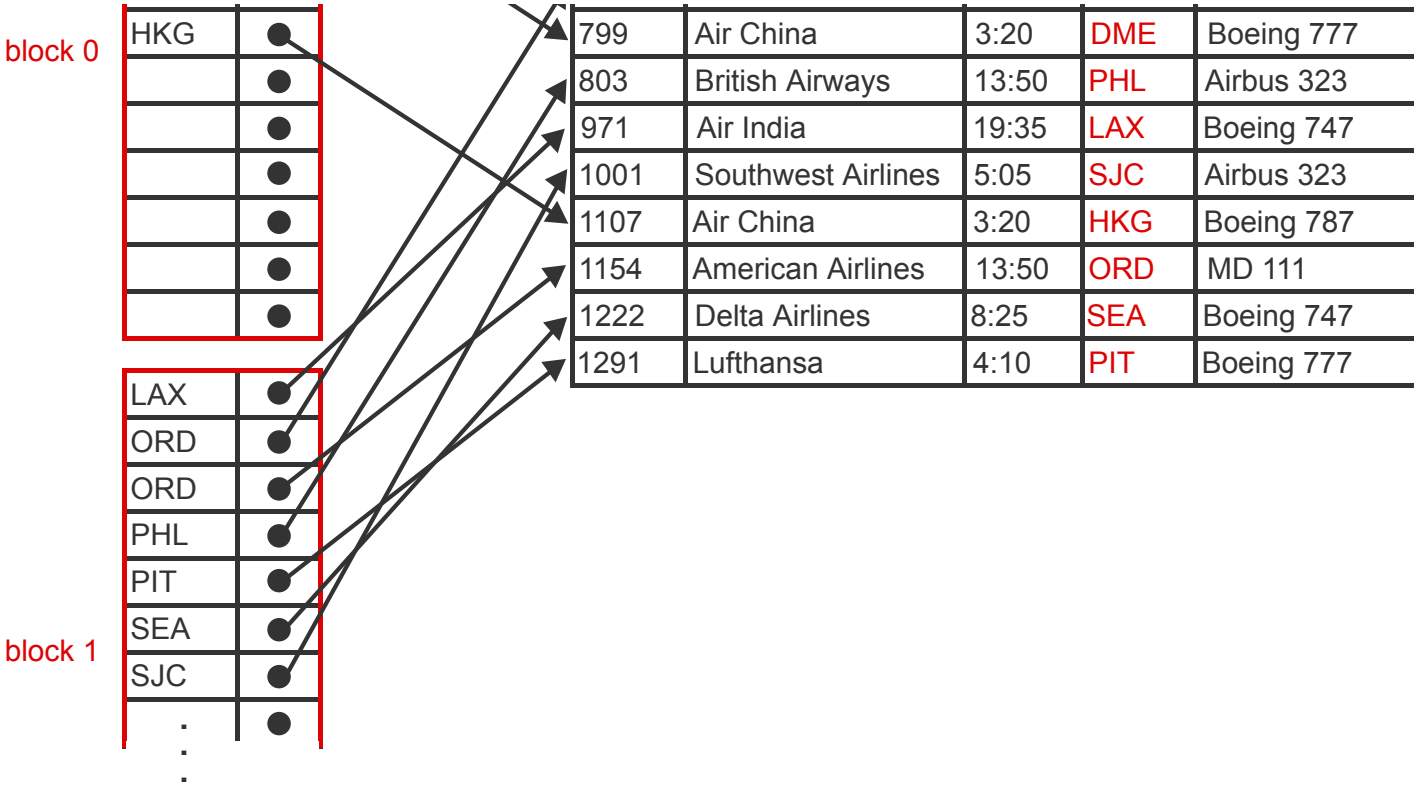
Inserts, updates, and deletes to tables have an impact on single-level indexes. Consider the behavior of dense indexes:

- *Insert*. When a row is inserted into a table, a new index entry is created. Since single-level indexes are sorted, the new entry must be placed in the correct location. To make space for the new entry, subsequent entries must be moved, which is too slow for large tables. Instead, the database splits an index block and reallocates entries to the new block, creating space for the new entry.
- *Delete*. When a row is deleted, the row's index entry must be deleted. The deleted entry can be either physically removed or marked as 'deleted'. Since single-level indexes are sorted, physically removing an entry requires moving all subsequent entries, which is slow. For this reason, index entries are marked as 'deleted'. Periodically, the database may reorganize the index to remove deleted entries and compress the index.
- *Update*. An update to a column that is not indexed does not affect the index. An update to an indexed column is like a delete followed by an insert. The index entry for the initial value is deleted and an index entry for the updated value is inserted.

With a sparse index, each entry corresponds to a table block rather than a table row. Index entries are inserted or deleted when blocks split or merge. Since blocks contain many rows, block splits and mergers occur less often than row inserts and deletes. Aside from frequency, however, the behavior of sparse and dense indexes is similar.

PARTICIPATION ACTIVITY 14.2.9: Insert with dense sorted index.





Animation content:

Step 1: A new row is inserted into the table with a dense sorted index. A new index entry 'PIT' must be inserted in sort order. There is a dense sorted index and a table. Every entry in the dense sorted index points to a row of the table. A new entry is added to the table. The value PIT appears between PHL and SEA in the index.

Step 2: Since the index block is full, the block splits. Half of the existing entries are moved to the new block. Because the index block is full, it splits into blocks 0 and 1.

Step 3: The block split creates space for the new index entry. Index entry PIT is inserted in block 1 between PHL and SEA.

Animation captions:

1. A new row is inserted into the table with a dense sorted index. A new index entry 'PIT' must be inserted in sort order.
2. Since the index block is full, the block splits. Half of the existing entries are moved to the new block.
3. The block split creates space for the new index entry.

PARTICIPATION ACTIVITY 14.2.10: Inserts, updates, and deletes.

1) Inserts to a sorted table are always faster when the table has no indexes.

- True
- False

2) Inserts to a heap table are always faster when the table has no secondary indexes.

- True
- False

3) Most large tables have a primary index.

- True
- False

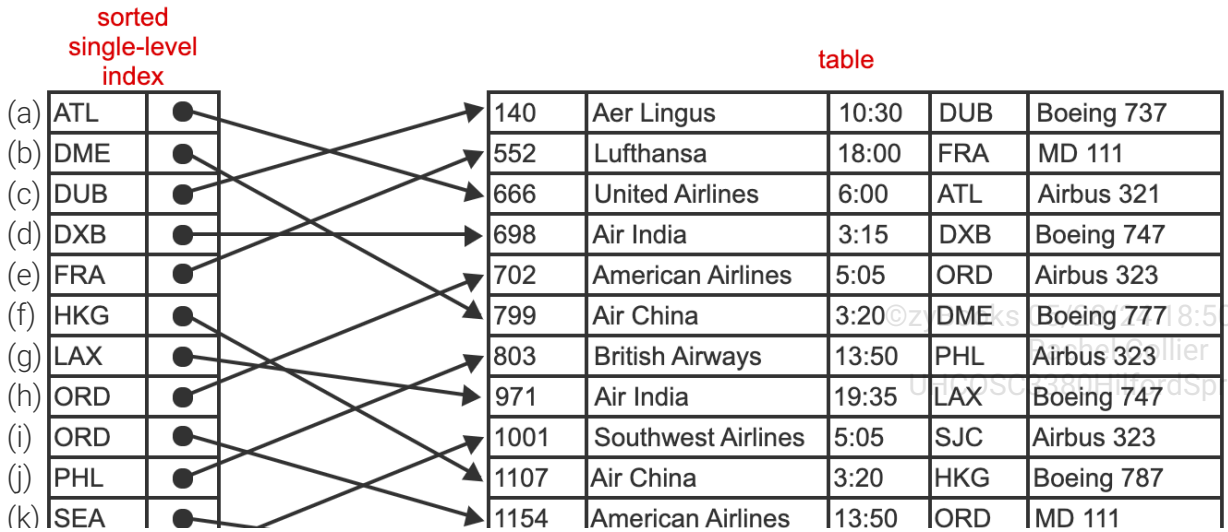
4) A table update may cause an index block split.

- True
- False

CHALLENGE ACTIVITY | 14.2.1: Single-level indexes.

544874.3500394.qx3zqy7

Start



Select the index entry that refers to the row for American Airlines flight 1154.

Pick

1 2 3 4 5 6

Check Next

14.3 LAB - Create index and explain (Sakila)

This lab illustrates the use of indexes and EXPLAIN to optimize query performance. Refer to EXPLAIN documentation for information about EXPLAIN result columns.

Refer to the `film` table of the Sakila database. Write and run seven SQL statements:

1. Explain the query `SELECT * FROM film WHERE title = 'ALONE TRIP';`.

   *In the EXPLAIN result, column **`key`** is null, indicating no index is available for the query. Column **`rows`** is 100, indicating all rows are read. The query executes a table scan.*

2. Create an index `idx_title` on the `title` column.
3. Explain the query of step 1 again.

   *In the EXPLAIN result, column **`key`** has value **`idx_title`**, indicating the query uses the index on **`title`**. Column **`rows`** is 1, indicating only one table row is read. The query executes an index scan, which is faster that the table scan of step 1.*

4. Explain the query `SELECT * FROM film WHERE title > 'ALONE TRIP';`.

   *In the EXPLAIN result, column **`key`** is null, indicating the query does not use the **`idx_title`** index. Column **`rows`** is 100, indicating all rows are read. The query ignores the index and executes a table scan.*

5. Explain the query `SELECT rating, count(*) FROM film GROUP BY rating;`

   *In the EXPLAIN result, column **`key`** is null, indicating no index is available for the query. Column **`rows`** is 100, indicating all rows are read. The query executes a table scan.*

6. Create an index `idx_rating` on the `rating` column.
7. Explain the query of step 5 again.

*In the EXPLAIN result, column **`key`** has value **`idx_rating`**, indicating the query reads **`rating`** values from the index. The query executes an index scan, which is faster than the table scan of step 5.*

NOTES:

For submit-mode testing, all seven statements must appear in the correct order. Submit-mode tests display multiple result tables as one table.

`SELECT * FROM film;` generates too many characters to display in the zyLab environment. However, statements with less output, such as `SELECT title FROM film;`, execute successfully.

To run this lab in the Sakila database of MySQL Workbench, drop the index `idx_title` from `film` prior to executing statement 1.

544874.3500394.qx3zqy7

LAB ACTIVITY | 14.3.1: LAB - Create index and explain (Sakila) | 10 / 10

Main.sql — Load default template...

```
1  -- Explain the query SELECT * FROM film WHERE title = 'ALONE TRIP';
2  EXPLAIN SELECT * FROM film WHERE title = 'ALONE TRIP';
3
4  -- Create an index idx_title on the title column.
5  CREATE INDEX idx_title ON film (title);
6
7  -- Explain the query of step 1 again.
8  EXPLAIN SELECT * FROM film WHERE title = 'ALONE TRIP';
9
10 -- Explain the query SELECT * FROM film WHERE title > 'ALONE TRIP';
11 EXPLAIN SELECT * FROM film WHERE title > 'ALONE TRIP';
12
13 -- Explain the query SELECT rating, count(*) FROM film GROUP BY rating;
14 EXPLAIN SELECT rating, count(*) FROM film GROUP BY rating;
15
```

Develop mode | Submit mode

Explore the database and run your program as often as you'd like, before submitting for grading. Click **Run program** and observe the program's output in the second box.

Run program | Main.sql → Output (shown below)

(Your program)

Program output displayed here

Coding trail of your work [What is this?](#)

```
4/3 W0,0,0,10 min:3
```

14.4 LAB - Query execution plans (Sakila)

This lab illustrates how minor changes in a query may have a significant impact on the execution plan.

MySQL Workbench exercise

Refer to the `film`, `actor`, and `film_actor` tables of the Sakila database. This exercise is based on the initial Sakila installation. If you have altered these tables or their data, your results may be different.

Do the following in MySQL Workbench:

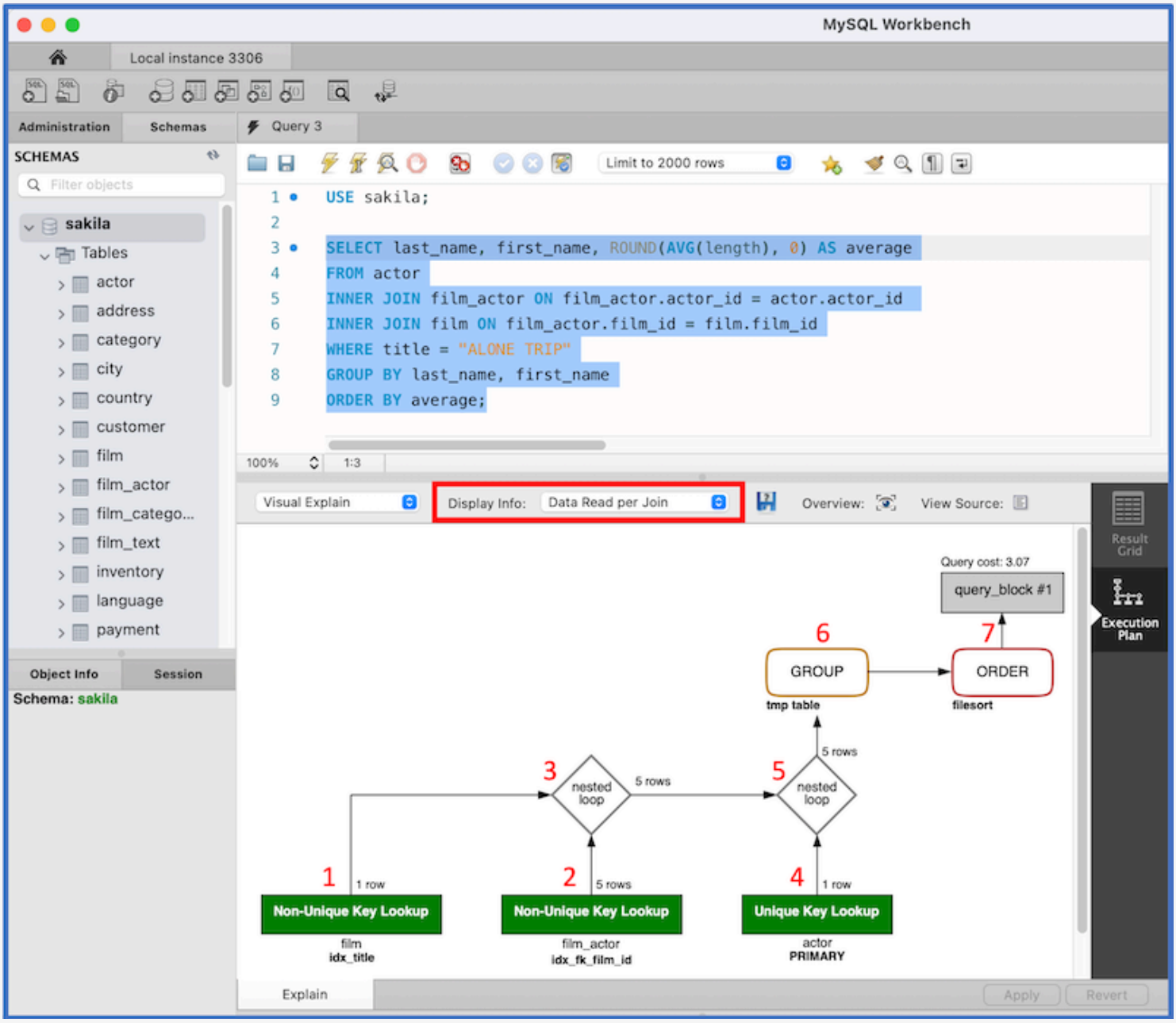
1. Enter the following statements:

```
USE sakila;

SELECT last_name, first_name, ROUND(AVG(length), 0) AS average
FROM actor
INNER JOIN film_actor ON film_actor.actor_id = actor.actor_id
INNER JOIN film ON film_actor.film_id = film.film_id
WHERE title = "ALONE TRIP"
GROUP BY last_name, first_name
ORDER BY average;
```

2. Highlight the SELECT query.
3. In the main menu, select Query > Explain Current Statement.
4. In the Display Info box, highlighted in red below, select Data Read per Join

Workbench displays the following execution plan:



The execution plan depicts the result of EXPLAIN for the SELECT query. The execution plan has seven steps, corresponding to the red numbers on the screenshot:

1. Access a single `film` row using the `idx_title` index on the `title` column.
2. Access matching `film_actor` rows using the `idx_fk_film_id` index on the `film_id` foreign key.
3. Join the results using the nested loop algorithm.
4. Access `actor` rows via the index on the primary key.
5. Join `actor` rows with the prior join result using the nested loop algorithm.
6. Store the result in a temporary table and compute the aggregate function.
7. Sort and generate the result table.

Refer to MySQL nested loop documentation for an explanation of the nested loop algorithm.

Now, replace = in the WHERE clause with < and generate a new execution plan. Step 1 of the execution plan says Index Range Scan. The index scan accesses all films with titles preceding "ALONE TRIP", rather than a single film.

Finally, replace < in the WHERE clause with > and generate a third execution plan. Step 1 of the execution plan says Full Table Scan and accesses `actor` rather than `film`.

zyLab coding

In the zyLab environment, write EXPLAIN statements for the three queries, in the order described above. Submit the EXPLAIN statements for testing.

The zyLab execution plans do not exactly match the Workbench execution plans, since this lab uses a subset of `film`, `actor`, and `film_actor` rows from the Sakila database.

NOTE: In submit-mode tests that generate multiple result tables, the results are merged. Although the tests run correctly, the results appear in one table.

544874.3500394.qx3zqy7

LAB ACTIVITY | 14.4.1: LAB - Query execution plans (Sakila) | 10 / 10

Main.sql — Load default template...

```
1  -- Your EXPLAIN statements go here
2
3  -- First execution plan: WHERE clause using =
4  EXPLAIN SELECT last_name, first_name, ROUND(AVG(length), 0) AS average
5  FROM actor
6  INNER JOIN film_actor ON film_actor.actor_id = actor.actor_id
7  INNER JOIN film ON film_actor.film_id = film.film_id
8  WHERE title = "ALONE TRIP"
9  GROUP BY last_name, first_name
10 ORDER BY average;
11
12 -- Second execution plan: WHERE clause using <
13 EXPLAIN SELECT last_name, first_name, ROUND(AVG(length), 0) AS average
14 FROM actor
15 INNER JOIN film_actor ON film_actor.actor_id = actor.actor_id
```

Develop mode | Submit mode

Explore the database and run your program as often as you'd like, before submitting for grading. Click **Run**

program and observe the program's output in the second box.

Run program

Main.sql
(Your program)

→ Output (shown below)

Program output displayed here

Coding trail of your work [What is this?](#)

```
4/3 W0,0,10 min:2
```