38.1 Storage media

Storage media

Computers use a variety of media to store data, such as random-access memory, magnetic disk, optical disk, and magnetic tape. Computer media vary on four important dimensions:

- *Speed*. Speed is measured as access time and transfer rate. ***Access time*** is the time required to access the first byte in a read or write operation. ***Transfer rate*** is the speed at which data is read or written, following initial access.
- *Cost*. Cost typically ranges from pennies to dollars per gigabyte of memory, depending on the media type.
- *Capacity*. In principle, any media type can store any amount of data. In practice, capacity is limited by cost. Expensive media have limited capacity compared to inexpensive media.
- *Volatility*. ***Volatile memory*** is memory that is lost when disconnected from power. ***Non-volatile memory*** is retained without power.

Three types of media are most important for database management:

- ***Main memory***, also called ***random-access memory (RAM)***, is the primary memory used when computer programs execute. Main memory is fast, expensive, and has limited capacity.
- ***Flash memory***, also called ***solid-state drive (SSD)***, is less expensive and higher capacity than main memory. Writes to flash memory are much slower than reads, and both are much slower than main memory writes and reads.
- ***Magnetic disk***, also called ***hard-disk drive (HDD)***, is used to store large amounts of data. Magnetic disk is slower, less expensive, and higher capacity than flash memory.

Main memory is volatile. Flash memory and magnetic disk are non-volatile and therefore considered storage media. Databases store data permanently on storage media and transfer data to and from main memory during program execution.

Table 38.1.1: Media characteristics (circa 2018).

Media type	Access time (microseconds)	Transfer rate (gigabyte/second)	Cost ($/gigabyte)	Volatile
Main memory	.01 to .1	> 10	> 1	Yes

Flash memory	20 to 100	.5 to 3	around .25	No
Magnetic disk	5,000 to 10,000	.05 to .2	around .02	No

PARTICIPATION ACTIVITY 38.1.1: Storage media.

Match the media type to the descriptive phrase.

If unable to drag and drop, refresh the page.

Flash memory **Main memory** **Magnetic disk**

	Reading one gigabyte takes about one second.
	Used to store petabytes (millions of gigabytes) of user data in the cloud.
	Upgrading from 16 to 32 gigabytes costs an extra $400 for an Apple laptop computer.

Reset

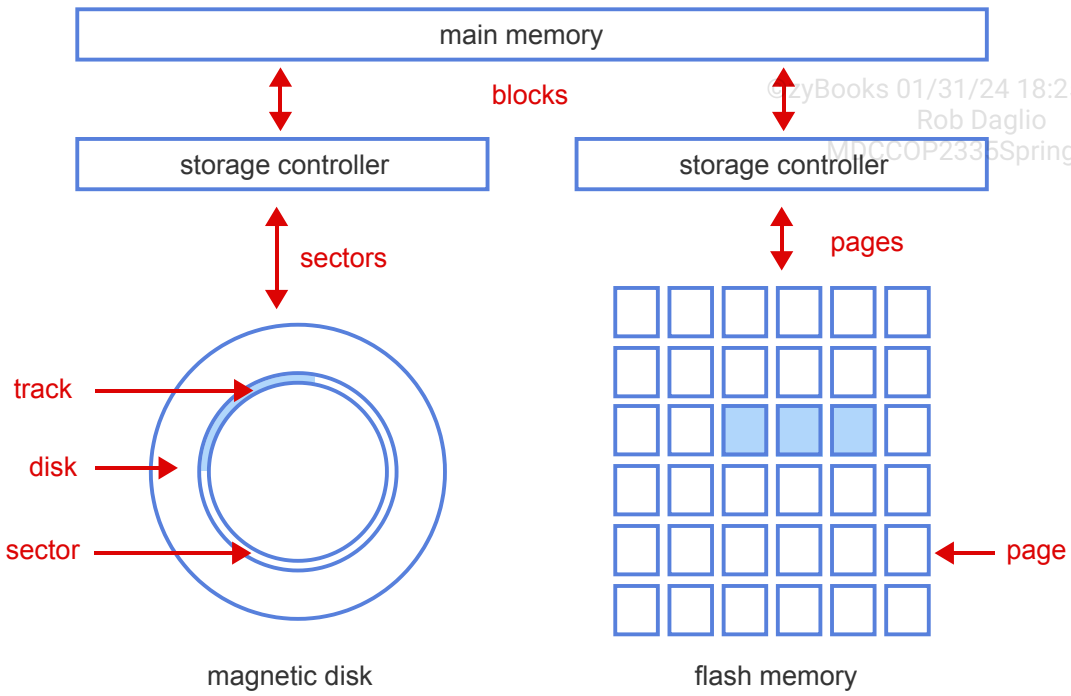
Sectors, pages, and blocks

Magnetic disk groups data in ***sectors***, traditionally 512 bytes per sector but 4 kilobytes with newer disk formats. Flash memory groups data in ***pages***, usually between 2 kilobytes and 16 kilobytes per page.

Databases and file systems use a uniform size, called a ***block***, when transferring data between main memory and storage media. Block size is independent of storage media. Storage media are managed by controllers, which convert between blocks and sectors or pages. This conversion is internal to the storage device, so the database is unaware of page and sector sizes.

Block size must be uniform within a database but, in most database systems, can be specified by the database administrator. Database systems typically support block sizes ranging from 2 kilobytes to 64 kilobytes. Smaller block sizes are usually better for transactional applications, which access a few rows per query. Larger block sizes are usually better for analytic applications, which access many rows per query.

PARTICIPATION ACTIVITY 38.1.2: Sectors, pages, and blocks.



Animation content:

Step 1: Magnetic disks store data in sectors. Each sector is a segment of a circular track. There is a circle labeled magnetic disk. There is a circle within the circle and labeled track. A section of the track is highlighted and labeled a sector.

Step 2: Flash memory stores data in pages. There is a 6x6 grid of squares labeled flash memory. One square is highlighted and labeled a page.

Step 3: Storage controllers convert blocks to pages on flash memory. There is a main memory that uses blocks and a storage controller that uses pages. There is a two way arrow between the main memory and storage controller and between the storage controller and the flash memory. A block moves from the main memory to the storage controller and then 3 pages leave the storage controller and enter the flash memory.

Step 4: Storage controllers convert blocks to sectors on magnetic disk. A new storage controller appears that uses sectors and a two way arrow appears between the main memory and the new storage controller and between the new storage controller and the magnetic disk. A block goes from the main memory to the storage controller and then two sectors go from the storage controller to the magnetic disk.

Animation captions:

1. Magnetic disks store data in sectors. Each sector is a segment of a circular track.
2. Flash memory stores data in pages.
3. Storage controllers convert blocks to pages on flash memory.
4. Storage controllers convert blocks to sectors on magnetic disk.

PARTICIPATION ACTIVITY 38.1.3: Sectors, pages, and blocks.

1) Approximately how many sectors are required to store one megabyte of data?

- ○ Always 8,000
- ○ Always 2,000
- ○ Either 250 or 2,000

2) The database administrator specifies an eight-kilobyte block size. Flash memory page size is two kilobytes. A user runs a query that reads two pages of flash memory. How many blocks are transferred to main memory?

- ○ One-half
- ○ One
- ○ Four

Row-oriented storage

Accessing storage media is relatively slow. Since data is transferred to and from storage media in blocks, databases attempt to minimize the number of blocks required for common queries.

Most relational databases are optimized for transactional applications, which often read and write individual rows. To minimize block transfers, relational databases usually store an entire row within one block, which is called ***row-oriented storage***.

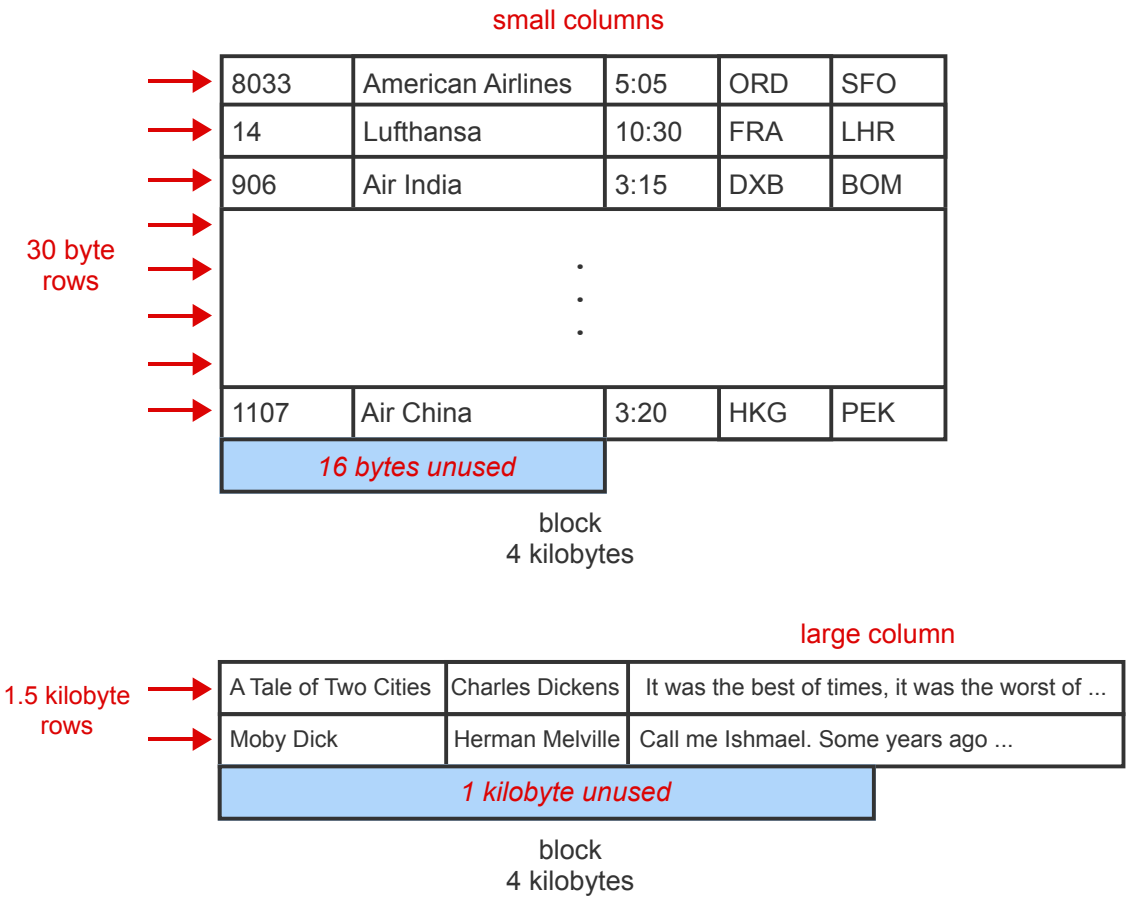
Row-oriented storage performs best when row size is small relative to block size, for two reasons:

- *Improved query performance.* When row size is small relative to block size, each block contains many rows. Queries that read and write multiple rows transfer fewer blocks, resulting in better performance.
- *Less wasted storage.* Row-oriented storage wastes a few bytes per block, since rows do not usually fit evenly into the available space. The wasted space is less than the row size. If row size is small relative to block size, this wasted space is insignificant.

Consequently, database administrators might specify a larger block size for databases containing larger rows.

Sometimes a table contains a very large column, such as 1 megabyte documents or 10 megabyte images. For tables with large columns, each row usually contains a link to the large column, which is stored in a different area. The large column might be stored in files managed by the operating system or in a special storage area managed by the database. This approach keeps row size small and improves performance of queries that do not access the large column.

PARTICIPATION ACTIVITY 38.1.4: Row-oriented storage.



Animation content:

Step 1: The row size is small relative to the block size, so many rows are transferred per block. A 4 kilobyte block of data appears with caption small columns. The block contains many data rows and one empty row. Each data row includes flight number, airline name, departure time, departure airport code, and arrival airport code.

Step 2: 4 kilobytes - (30 bytes/row × 136 rows) = 4096 bytes - 4080 bytes = 16 bytes unused. So

wasted space is insignificant. The caption 30 byte rows appears with arrows pointing to every data row. The empty row is labeled 16 bytes unused.

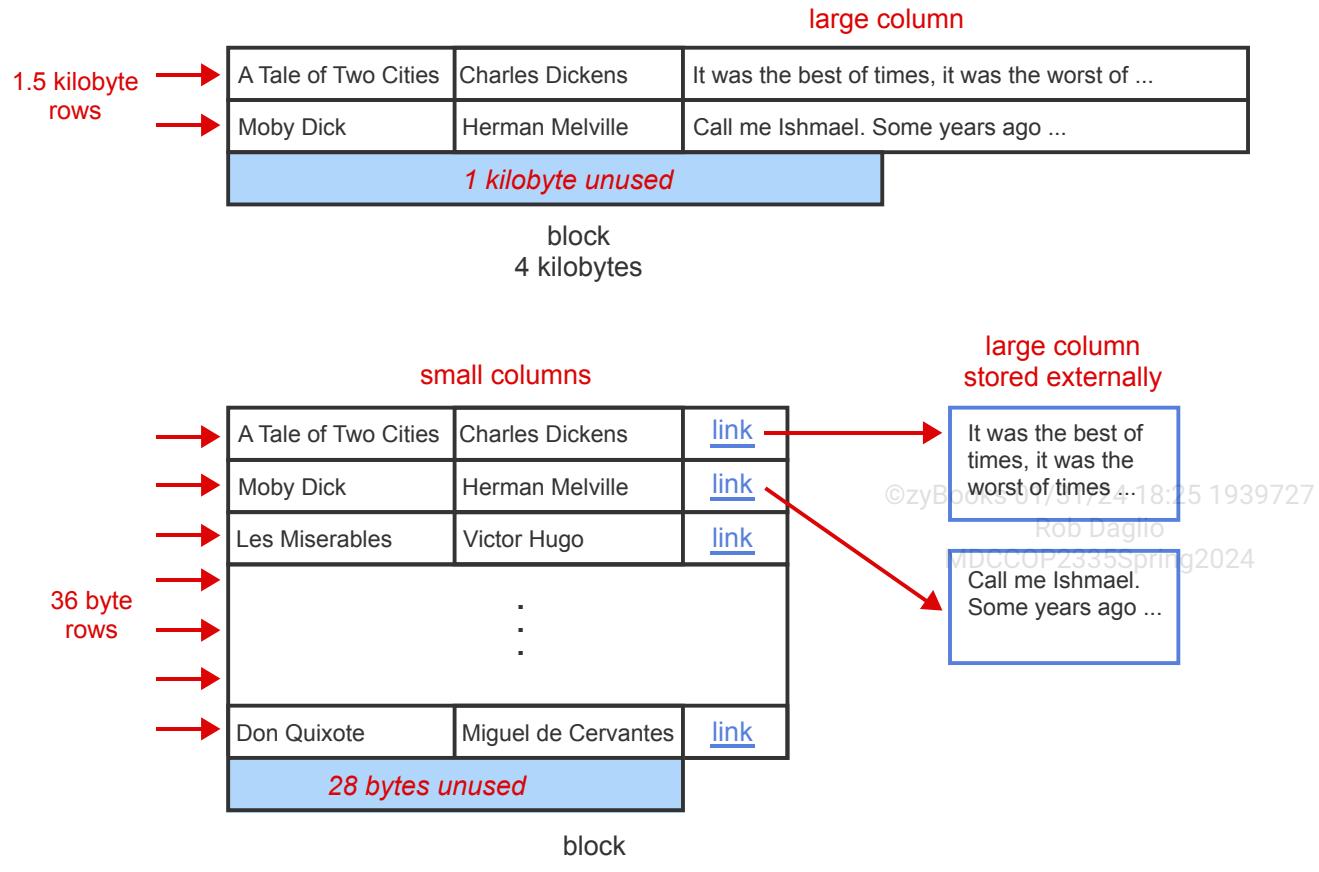
Step 3: The row size is large relative to block size, so fewer rows are transferred per block. Another 4 kilobyte block appears with two data rows and one empty row. Each data row contains book title, book author, and book text. The book text column has caption large column.

Step 4: 4 kilobytes - (1.5 kilobytes/row × 2 rows) = 1 kilobyte = 1024 bytes unused. So wasted space is significant. Next to the second block, the caption 1.5 kilobyte rows appears with arrows pointing to the data rows. The empty row is labeled 1 kilobyte unused.

Animation captions:

1. The row size is small relative to the block size, so many rows are transferred per block.
2. 4 kilobytes - (30 bytes/row × 136 rows) = 4096 bytes - 4080 bytes = 16 bytes unused. So wasted space is insignificant.
3. The row size is large relative to block size, so fewer rows are transferred per block.
4. 4 kilobytes - (1.5 kilobytes/row × 2 rows) = 1 kilobyte = 1024 bytes unused. So wasted space is significant.

PARTICIPATION ACTIVITY 38.1.5: Row-oriented storage with large columns.



4 kilobytes

Animation content:

Step 1: The large column allows only two rows to be transferred per block and wastes significant space. There is a 4 kilobyte block with two rows of 1.5 kilobytes with a large column and has 1 kilobyte unused.

Step 2: Large columns are replaced by a link and are stored in a separate area. There is a new kilobyte block with two small columns. The large column is replaced by links and has been stored externally. This new 4 kilobyte block has more rows than the first 4 kilobyte block.

Step 3: More rows fit per block, and less space is wasted. Queries that do not access the large column are faster. Each row in the new block is 36 bytes and has 28 bytes unused.

Animation captions:

1. The large column allows only two rows to be transferred per block and wastes significant space.
2. Large columns are replaced by a link and are stored in a separate area.
3. More rows fit per block, and less space is wasted. Queries that do not access the large column are faster.

PARTICIPATION ACTIVITY 38.1.6: Row-oriented storage.

1) A database uses 16-kilobyte blocks. A table contains 18,200 rows of 100 bytes each. Ignoring space used by the system for overhead, how many bytes are unused on each block? Assume one kilobyte = 1,024 bytes.

   - None
   - 84
   - 384

2) 4-megabyte photographs are usually stored ____.

- in the same blocks as other columns of the table
- separate from other columns, managed by the file system
- separate from other columns, managed either within the database or by the file system

Column-oriented storage

Some newer relational databases are optimized for analytic applications rather than transactional applications. Analytic applications often read just a few columns from many rows. In this case, column-oriented storage is optimal. In ***column-oriented*** storage, also called ***columnar storage***, each block stores values for a single column only.

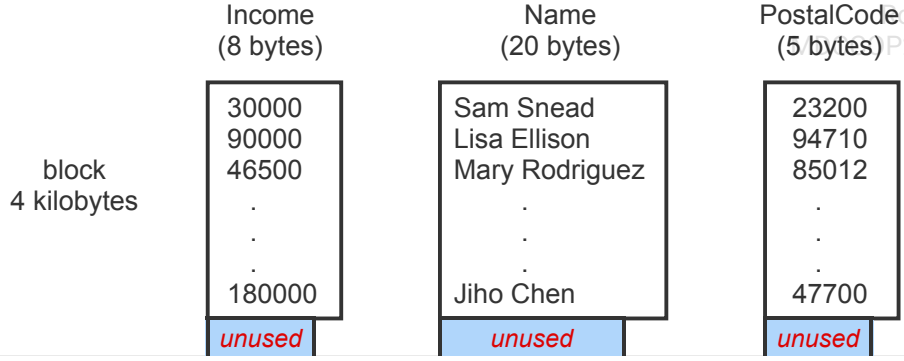
Column-oriented storage benefits analytic applications in several ways:

- *Faster data access*. More column values are transferred per block, reducing time to access storage media.
- *Better data compression*. Databases often apply data compression algorithms when storing data. Data compression is usually more effective when all values have the same data type. As a result, more values are stored per block, which reduces storage and access time.

With column-oriented storage, reading or writing an entire row requires accessing multiple blocks. Consequently, column-oriented storage is a poor design for most transactional applications.

PostgreSQL and Vertica are examples of relational databases that support column-oriented storage. Many NoSQL databases, described elsewhere in this material, are optimized for analytic applications and use column-oriented storage.

PARTICIPATION ACTIVITY 38.1.7: Column-oriented storage.



```
SELECT AVG(Income)
FROM Person
```

AVG(Income)
52844

```
SELECT Name, PostalCode, Income
FROM Person
WHERE PostalCode = 23200
```

Name	PostalCode	Income
Sam Snead	23200	30000

Animation content:

Step 1: With column-oriented storage, a 4 kilobyte block contains up to 512 8-byte column values. There is a 4 kilobyte block with one column Income that takes up 8 bytes and the rest of the space is unused.

Step 2: Computing the average of all incomes reads fewer blocks than row-oriented storage. SQL code appears and states SELECT AVG left parenthesis Income right parenthesis. FROM Person. The average income is 52844.

Step 3: Different columns occupy separate blocks. Two more columns named Name and PostalCode appear with Name taking up 20 bytes and Postal Code taking up 5 bytes.

Step 4: Selecting multiple columns reads multiple blocks and is slower than row-oriented storage. New SQL code appears and states SELECT Name comma PostalCode comma Income. FROM Person. WHERE PostalCode = 23200. A new table appears with columns Name, PostalCode, and Income with one row with the values Sam Snead 23200 and 30000.

Animation captions:

1. With column-oriented storage, a 4 kilobyte block contains up to 512 8-byte column values.
2. Computing the average of all incomes reads fewer blocks than row-oriented storage.
3. Different columns occupy separate blocks.
4. Selecting multiple columns reads multiple blocks and is slower than row-oriented storage.

Terminology

*In this material, the terms **column-oriented** and **columnar** refer to a data storage technique. Occasionally, these terms refer to a type of NoSQL database, commonly called a **wide column database** and described elsewhere in this material.*

PARTICIPATION ACTIVITY 38.1.8: Column-oriented storage.

Refer to the animation above. Assume the database contains 1,000,000 people.

1) Assume column-oriented storage. Roughly how many blocks are required to compute average income by postal code for all people?
   - 2,000 blocks
   - 3,250 blocks
   - The number of blocks depends on row size.

2) Assume row-oriented storage, with each row containing Name, Income, and PostalCode only. Roughly how many blocks are required to compute average income by postal code for all people?
   - 3,250 blocks
   - 4,133 blocks
   - 8,264 blocks

3) The term 'column-oriented' ____.
   - always means a technique for organizing data on storage media
   - always means queries that specify multiple columns in the SELECT clause
   - means either a technique for organizing data on storage media or a type of NoSQL database

Exploring further:

- PostgreSQL column-oriented storage

- Vertica column-oriented storage

38.2 Table structures

Heap table

Row-oriented storage performs better than column-oriented storage for most transactional databases. Consequently, relational databases commonly use row-oriented storage. A ***table structure*** is a scheme for organizing rows in blocks on storage media.

Databases commonly support four alternative table structures:

- Heap table
- Sorted table
- Hash table
- Table cluster

Each table in a database can have a different structure. Databases assign a default structure to all tables. Database administrators can override the default structure to optimize performance for specific queries.

In a ***heap table***, no order is imposed on rows. The database maintains a list of blocks assigned to the table, along with the address of the first available space for inserts. If all blocks are full, the database allocates a new block and inserts rows in the new block.

When a row is deleted, the space occupied by the row is marked as free. Typically, free space is tracked as a linked list, as in the animation below. Inserts are stored in the first space in the list, and the head of the list is set to the next space.

Heap tables optimize insert operations. Heap tables are particularly fast for bulk load of many rows, since rows are stored in load order. Heap tables are not optimal for queries that read rows in a specific order, such as a range of primary key values, since rows are scattered randomly across storage media.

PARTICIPATION ACTIVITY 38.2.1: Heap table.

Animation content:

Step 1: In a heap table, the database maintains a pointer to free space, which indicates the location of the next insert. There is a table with five columns and has an end of table and a free space pointer to the end of the table.

Step 2: When a row is inserted, the pointer moves to the next available space. A new row is added to the end of the table so the free space pointer must move to the new end of the table.

Step 3: When a row is deleted, the pointer moves to the deleted space. The deleted space is linked to free space at the end of the table. A row is deleted in the table and the free space pointer moves to the deleted space. This new deleted space is linked to the end of the table.

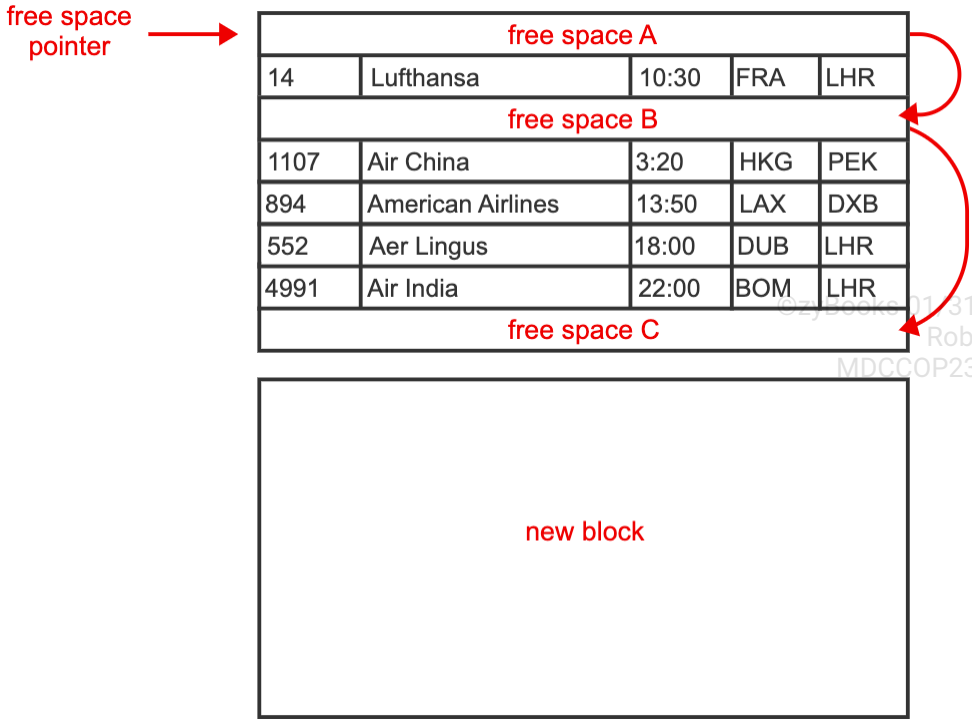
Step 4: As more rows are deleted, free space is linked together in a list. Another row is deleted that comes before in the table. The free space pointer moves to this deleted space and the deleted space is linked to the previously deleted row.

Step 5: Inserts go to the first available space in the list. A new entry is added and it goes to the space with the free space pointer. The information is added and the free space pointer is moved to the other deleted row.

Animation captions:

1. In a heap table, the database maintains a pointer to free space, which indicates the location of the next insert.
2. When a row is inserted, the pointer moves to the next available space.
3. When a row is deleted, the pointer moves to the deleted space. The deleted space is linked to free space at the end of the table.
4. As more rows are deleted, free space is linked together in a list.
5. Inserts go to the first available space in the list.

PARTICIPATION ACTIVITY 38.2.2: Heap table.



Refer to the heap table above. Four new rows are inserted into the table. Match the free space to the insert.

If unable to drag and drop, refresh the page.

Free space C | Free space B | Free space A | New block

	First insert
	Second insert
	Third insert
	Fourth insert

Reset

Sorted table

In a ***sorted table***, the database designer identifies a ***sort column*** that determines physical row order. The sort column is usually the primary key but can be a non-key column or group of columns.

Rows are assigned to blocks according to the value of the sort column. Each block contains all rows with values in a given range. Within each block, rows are located in order of sort column values.

Sorted tables are optimal for queries that read data in order of the sort column, such as:

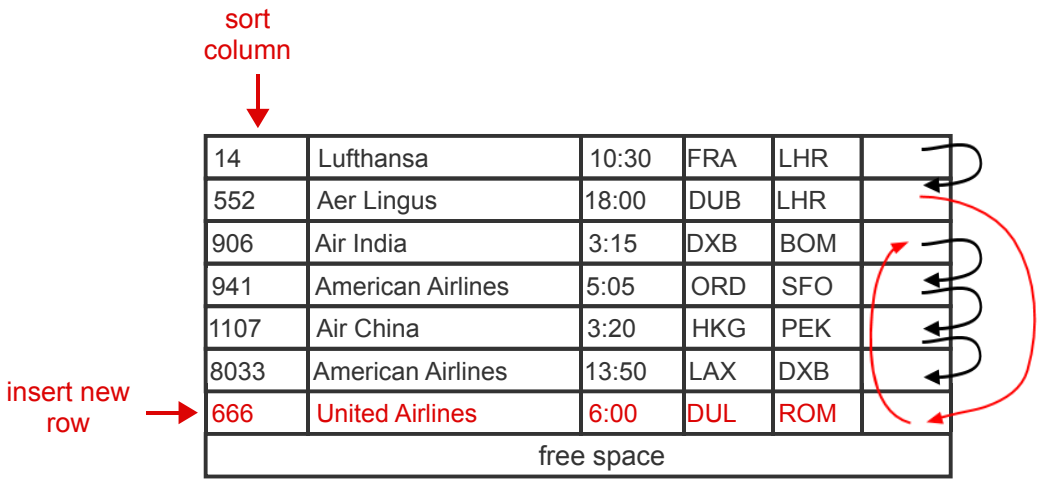
- JOIN on the sort column
- SELECT with range of sort column values in the WHERE clause
- SELECT with ORDER BY the sort column

Maintaining correct sort order of rows within each block can be slow. When a new row is inserted or when the sort column of an existing row is updated, free space may not be available in the correct location. To maintain the correct order efficiently, databases maintain pointers to the next row within each block, as in the animation below. With this technique, inserts and updates change two address values rather than move entire rows.

When an attempt is made to insert a row into a full block, the block splits in two. The database moves half the rows from the initial block to a new block, creating space for the insert.

In summary, sorted tables are optimized for read queries at the expense of insert and update operations. Since reads are more frequent than updates and inserts in many databases, sorted tables are often used, usually with the primary key as the sort column.

PARTICIPATION ACTIVITY 38.2.3: Sorted table.



Animation content:

Step 1: In a sorted table, rows are sorted on a sort column. A block contains a table with 5 columns and six rows. The first column is the sort column.

Step 2: Inserting a new row requires moving all subsequent rows, which is inefficient. A new row is inserted in sorted order between rows one and two.

Step 3: To avoid inefficient inserts, the sort column order is maintained with links, producing a linked list. The inserted row disappears. A new column appears. In each row, an arrow in the new column

points to the next row.

Step 4: The sort order is maintained during an insert by changing two links rather than moving rows. The row that was previously between rows one and two is inserted at the end of the table. The arrow for row one points to the new row. The arrow for the new row points to row two.

Animation captions:

1. In a sorted table, rows are sorted on a sort column.
2. Inserting a new row requires moving all subsequent rows, which is inefficient.
3. To avoid inefficient inserts, the sort column order is maintained with links, producing a linked list.
4. The sort order is maintained during an insert by changing two links rather than moving rows.

PARTICIPATION ACTIVITY 38.2.4: Sorted table.

1) Assume 100 rows fit in each block of a sorted table. If a block is full and an insert causes the block to split, roughly what percentage of the new and old blocks is empty?
   - ○ 33%
   - ○ 50%
   - ○ 90%

2) A database designer creates a new Employee table with primary key 'EmployeeNumber', loads 1 million rows into the table, then runs many queries that join other tables on the EmployeeNumber column. What is the best structure for Employee?
   - ○ Sorted table
   - ○ Heap table
   - ○ Heap table initially, then restructure to sorted table after bulk load is complete

3) Sorted tables are always sorted on a single column.
   - ○ True
   - ○ False

Hash table

In a ***hash table***, rows are assigned to buckets. A ***bucket*** is a block or group of blocks containing rows. Initially, each bucket has one block. As a table grows, some buckets eventually fill up with rows, and the database allocates additional blocks. New blocks are linked to the initial block, and the bucket becomes a chain of linked blocks.

The bucket containing each row is determined by a hash function and a hash key. The ***hash key*** is a column or group of columns, usually the primary key. The ***hash function*** computes the bucket containing the row from the hash key.

Hash functions are designed to scramble row locations and evenly distribute rows across blocks. The ***modulo function*** is a simple hash function with four steps:

1. Convert the hash key by interpreting the key's bits as an integer value.
2. Divide the integer by the number of buckets.
3. Interpret the division remainder as the bucket number.
4. Convert the bucket number to the physical address of the block containing the row.

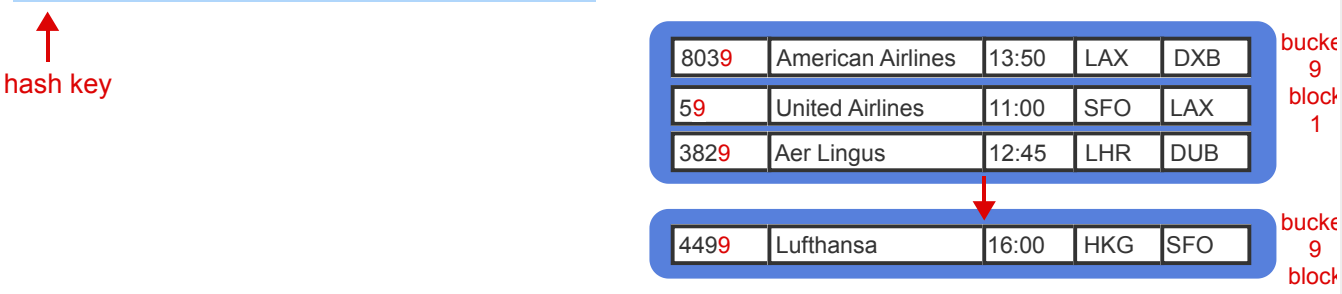
As tables grow, a fixed hash function allocates more rows to each bucket, creating deep buckets consisting of long chains of linked blocks. Deep buckets are inefficient since a query may read several blocks to access a single row. To avoid deep buckets, databases may use dynamic hash functions. A ***dynamic hash function*** automatically allocates more blocks to the table, creates additional buckets, and distributes rows across all buckets. With more buckets, fewer rows are assigned to each bucket and, on average, buckets contain fewer linked blocks.

Hash tables are optimal for inserts and deletes of individual rows, since row location is quickly determined from the hash key. For the same reason, hash tables are optimal for selecting a single row when the hash key value is specified in the WHERE clause. Hash tables are slow on queries that select many rows with a range of values, since rows are randomly distributed across many blocks.

PARTICIPATION ACTIVITY 38.2.5: Assigning rows to buckets with a modulo function.

Flight

Flight Number	AirlineName	Depart Time	Depart Airport	Arrive Airport
11	Lufthansa	10:30	FRA	LHR
552	Aer Lingus	18:00	DUB	LHR
908	Air India	3:15	DXB	BOM
940	American Airlines	5:05	ORD	SFO
1102	Air China	3:20	HKG	PEK
8039	American Airlines	13:50	LAX	DXB
59	United Airlines	11:00	SFO	LAX
3829	Aer Lingus	12:45	LHR	DUB
4499	Lufthansa	16:00	HKG	SFO



Animation content:

Step 1: FlightNumber is the hash key for the Flight table. Table Flight has columns FlightNumber, AirlineName, DepartTime, DepartAirport, and ArriveAirport. The first column is labeled hash key.

Step 2: The hash function is modulo 10, resulting in 10 buckets. Rows are assigned to buckets based on the last digit of the hash key. Buckets 0 to 9 appear and row 1 is added to bucket 1 because its flight number 11 ends in a 1. Row 2 i s added to bucket 2 because its flight number 552 ends in 2. All rows are added to their respective buckets.

Step 3: If a bucket fills with rows, additional blocks are allocated and linked to the first block. Three new rows with flight numbers that end in 9 are added. After bucket 9 is filled with three rows a new bucket 9 must be added. The old bucket is labeled block 1 and the new bucket is labeled block 2. The third new row is added to bucket 9 block 2.

Animation captions:

1. FlightNumber is the hash key for the Flight table.
2. The hash function is modulo 10, resulting in 10 buckets. Rows are assigned to buckets based on the last digit of the hash key.
3. If a bucket fills with rows, additional blocks are allocated and linked to the first block.

PARTICIPATION ACTIVITY 38.2.6: Hash table.

Assume blocks are 8 kilobytes, rows are 200 bytes, and the hash function is modulo 13.

1) If each bucket initially has one block, how many rows fit into a single bucket?

- 13
- 40
- 520

2) In a best-case scenario, what is the maximum number of row inserts before some bucket overflows?

- 40
- 520
- Unlimited

3) 1,000,000 rows are inserted. In a worst-case scenario, what is the maximum number of blocks chained together in a single bucket?

- 25,000
- 100
- 40

4) What happens when a new row is inserted into a full bucket?

- The full bucket splits in two, and half of the rows are moved to the new bucket.
- A new block is linked to the initial block, and half the rows are moved from the initial block to the new block.
- A new block is linked to the initial block, and the new row is stored in the new block.

CHALLENGE ACTIVITY 38.2.1: Table structures.

539740.3879454.qx3zqy7

Start

Refer to the heap table below. Free space A can accommodate two rows.

327	Santino	WY
347	Minenhle	MD
1551	Lois	KY
1897	Diego	CO

757	Yuna	PA
653	Asha	AR
free space A		

Where does the free space pointer point to?

Select

Row 757 is deleted and becomes free space B. What does the free space linked list look like?

Select

1 2 3 4 5

Check Next

Table clusters

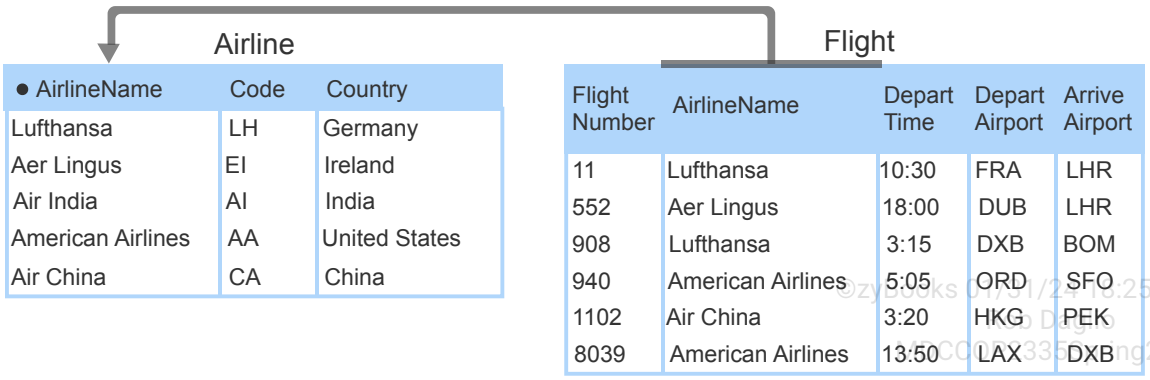
Table clusters, also called ***multi-tables***, interleave rows of two or more tables in the same storage area. Table clusters have a ***cluster key***, a column that is available in all interleaved tables. The cluster key determines the order in which rows are interleaved. Rows with the same cluster key value are stored together. Usually the cluster key is the primary key of one table and the corresponding foreign key of another, as in the animation below.

Table clusters are optimal when joining interleaved tables on the cluster key, since physical row location is the same as output order. Table clusters perform poorly for many other queries:

- *Join on columns other than cluster key*. In a join on a column that is not the cluster key, physical row location is not the same as output order, so the join is slow.
- *Read multiple rows of a single table*. Table clusters spread each table across more blocks than other structures. Queries that read multiple rows may access more blocks.
- *Update cluster key*. Rows may move to different blocks when the cluster key changes.

Table clusters are not optimal for many queries and therefore are not commonly used.

PARTICIPATION ACTIVITY 38.2.7: Table cluster.



Lufthansa	LH	Germany		
11	Lufthansa	10:30	FRA	LHR
908	Lufthansa	3:15	DXB	BOM
Aer Lingus	EI	Ireland		
552	Aer Lingus	18:00	DUB	LHR
Air India	AI	India		
American Airlines	AA	United States		
940	American Airlines	5:05	ORD	SFO
8039	American Airlines	13:50	LAX	DXB
Air China	CA	China		
1102	Air China	3:20	HKG	PEK

Animation content:

Static figure:
Two diagrams appear. The first diagram shows tables Airline and Flight. Table Airline has columns AirlineName, Code, and Country, with five rows. AirlineName is the primary key. Table Flight has columns FlightNumber, AirlineName, DepartTime, DepartAirport, and ArriveAirport, with six rows. FlightNumber is the primary key. An arrow points from AirlineName in table Flight to AirlineName in table Airline.

A second diagram represents data storage and shows interleaved rows of tables Airline and Flight. Each Airline row is followed by all Flight rows with the same AirlineName.

Step 1: Airline and Flight tables both contain an AirlineName column. The Airline and Flight tables appear. The AirlineName columns are highlighted.

Step 2: AirlineName is the cluster key for the table cluster. The AirlineName columns in both tables are labeled cluster keys.

Step 3: Rows of Airline and Flight are interleaved on storage media. The second diagram appears.

Animation captions:

1. Airline and Flight tables both contain an AirlineName column.
2. AirlineName is the cluster key for the table cluster.
3. Rows of Airline and Flight are interleaved on storage media.

PARTICIPATION ACTIVITY 38.2.8: Table clusters.

1) The cluster key is normally the primary key of both tables in a cluster.
   - True
   - False

2) Table clusters are the most commonly used physical structure for tables.
   - True
   - False

3) Cluster keys in a table cluster are similar to sort columns in a sorted table.
   - True
   - False

4) Table clusters always contain exactly two tables.
   - True
   - False

38.3 Single-level indexes

Single-level indexes

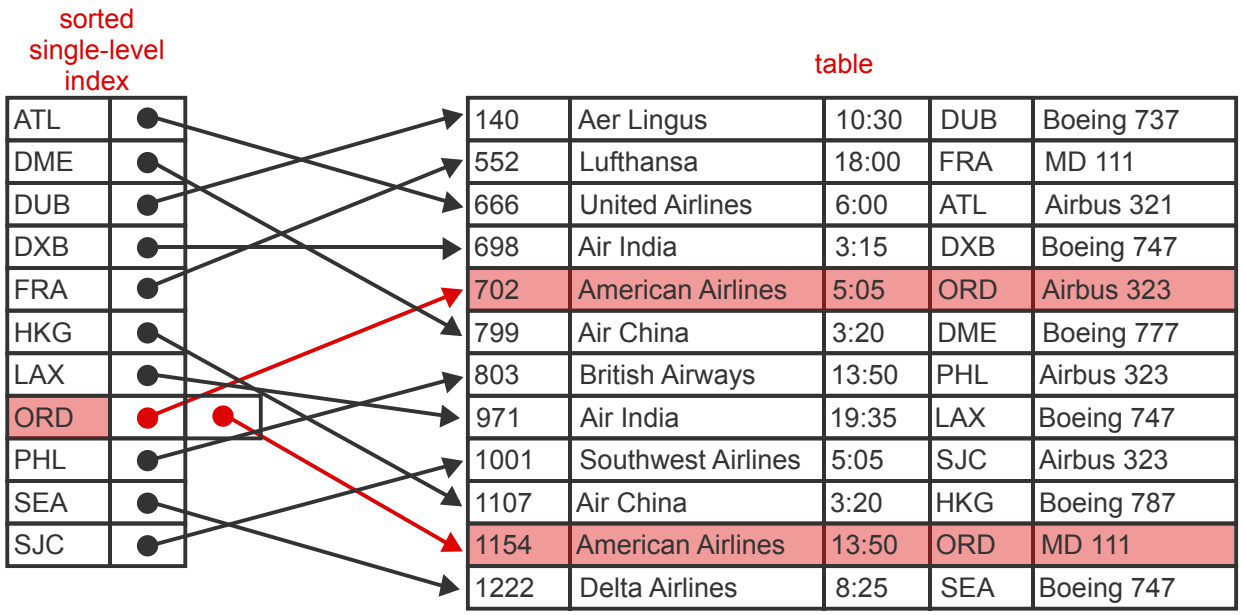
A ***single-level index*** is a file containing column values, along with pointers to rows containing the column value. The pointer identifies the block containing the row. In some indexes, the pointer also identifies the exact location of the row within the block. Indexes are created by database designers 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 38.3.1: Single-level index.



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 38.3.2: 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.

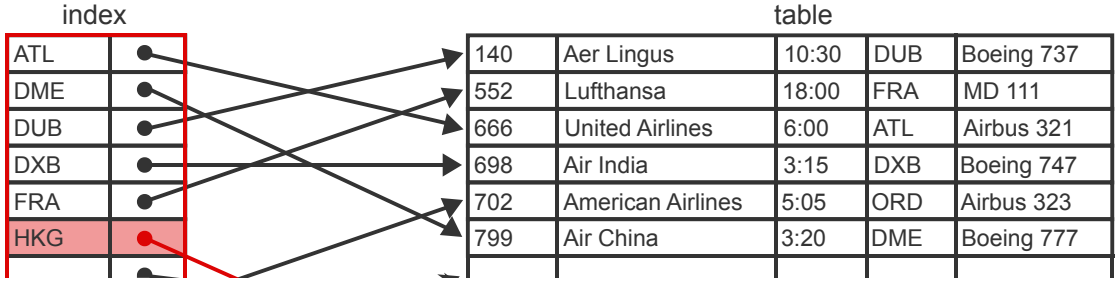
Hit ratio, also called ***filter factor*** or ***selectivity***, is the percentage of table rows selected by a query. When a SELECT query is executed, the database examines the WHERE clause and estimates hit ratio. If hit ratio is high, the database performs a table scan. If hit ratio is low, the query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

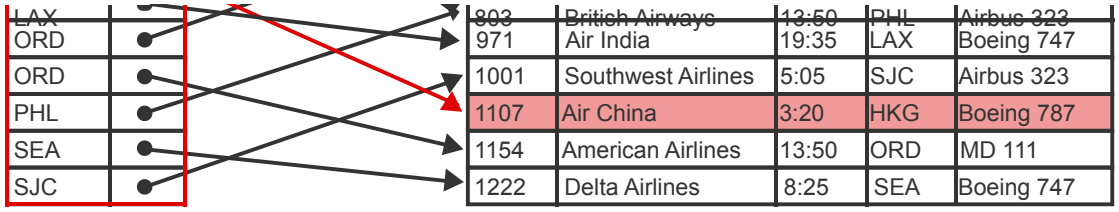
1. Looks for an indexed column in the WHERE clause.
2. Scans the index.
3. Finds values that match the WHERE clause.
4. Reads the corresponding table blocks.

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 38.3.3: 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:

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 38.3.4: 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".

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

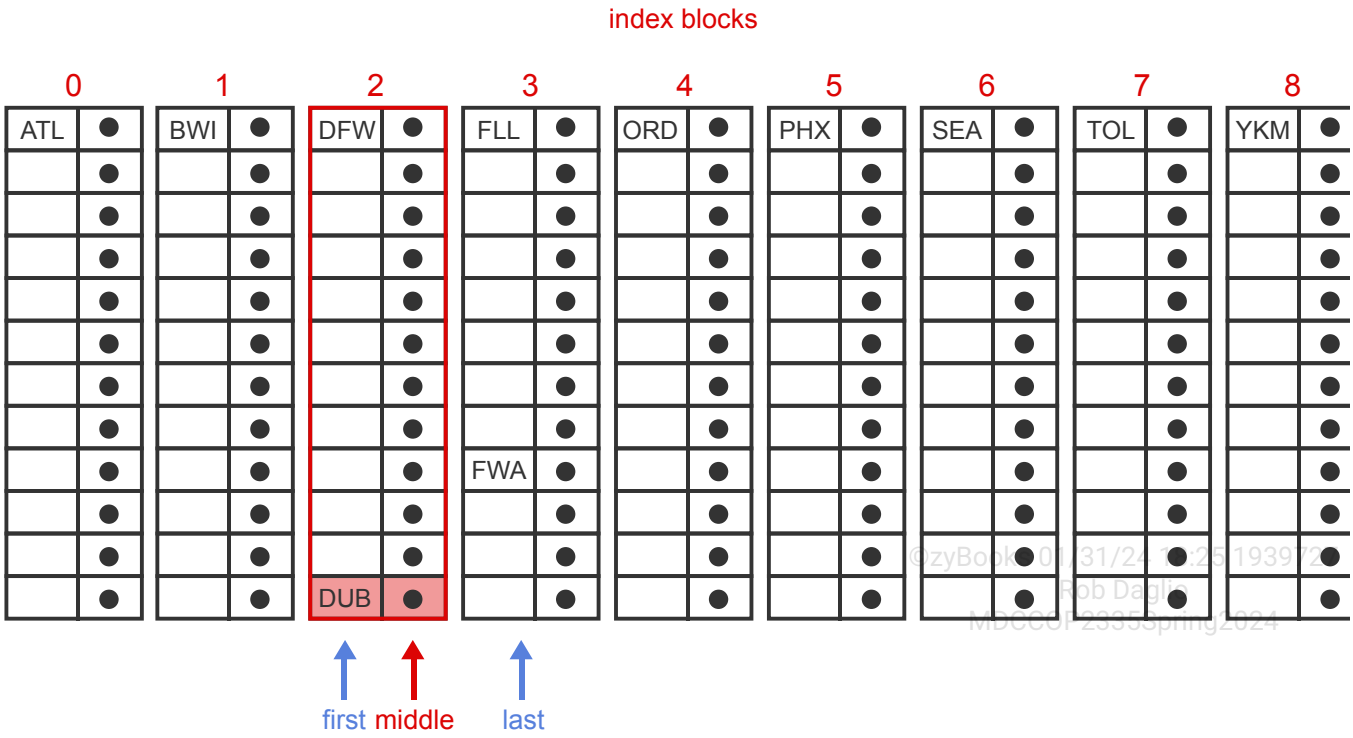
pointers to blocks containing rows selected by the query. When hit ratio is low, additional time to read table blocks is insignificant.

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 38.3.5: 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

38.3.6: 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.

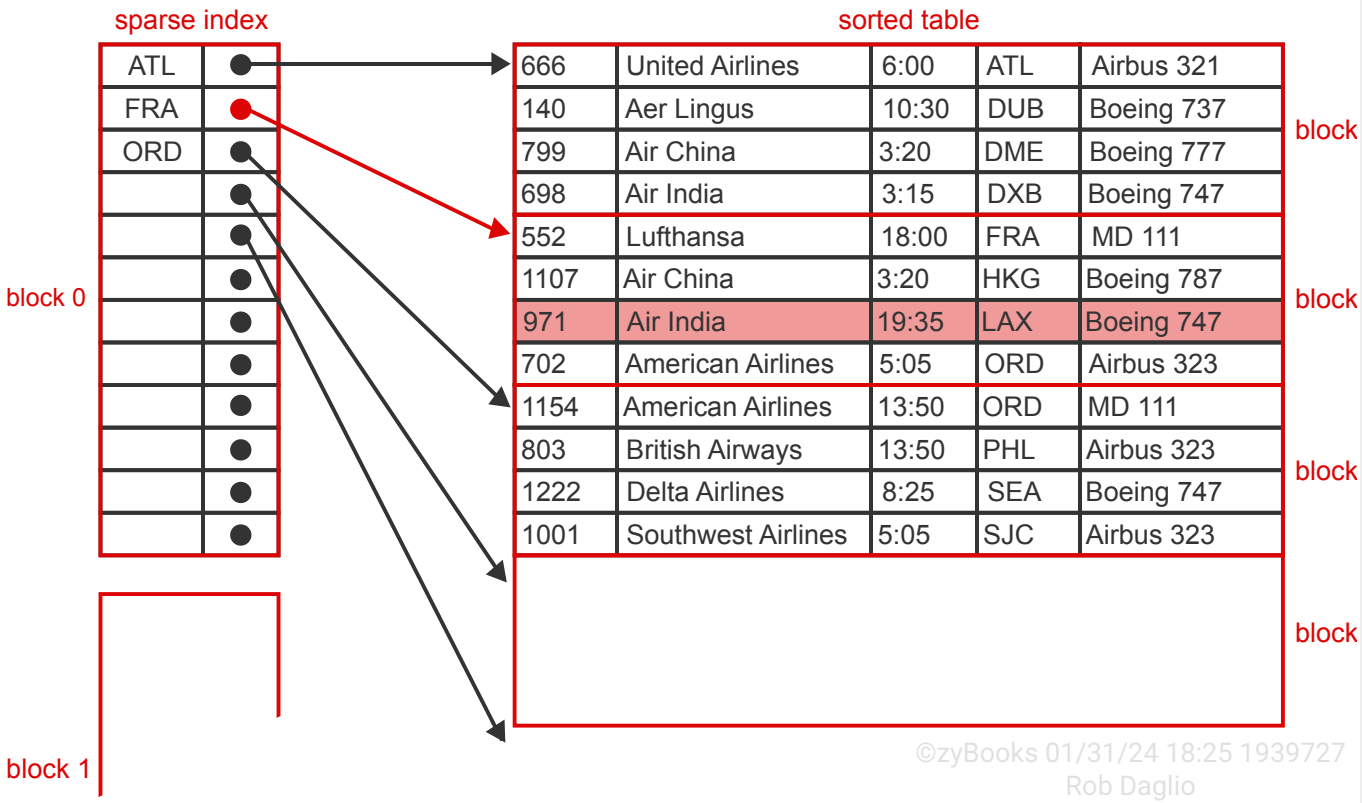
Sparse indexes are much faster than dense indexes since sparse indexes have fewer entries and occupy fewer blocks. Consider the following scenario:

- 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 38.3.7: 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 38.3.8: Primary and secondary indexes.

Match the term with the term's description.

If unable to drag and drop, refresh the page.

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

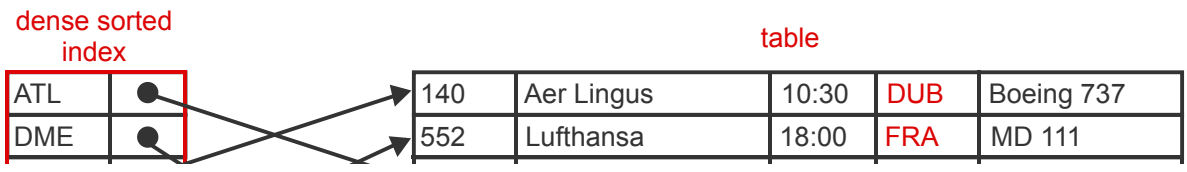
Inserts, updates, and deletes

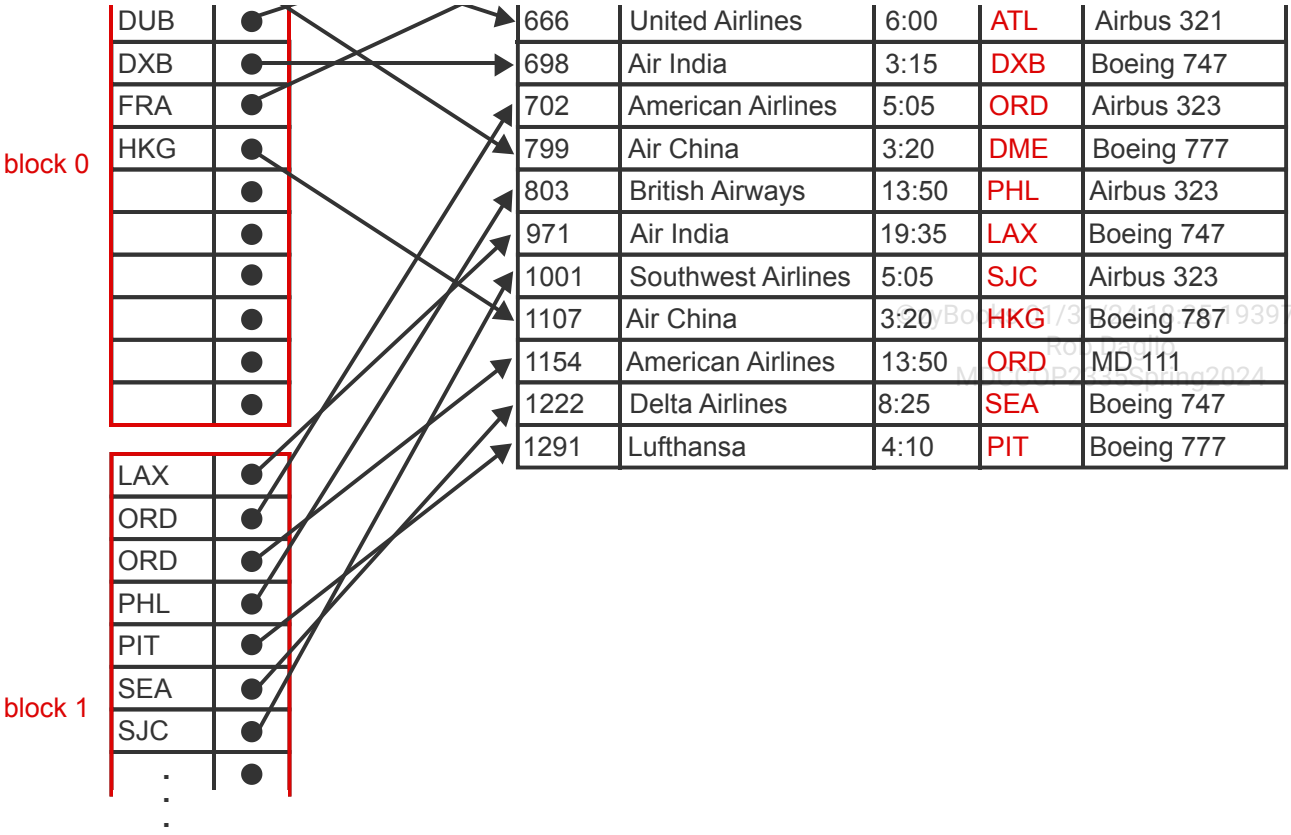
Inserts, updates, and deletes to tables have an impact on single-level indexes. Consider the behavior of dense indexes:

- *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 38.3.9: 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 38.3.10: 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

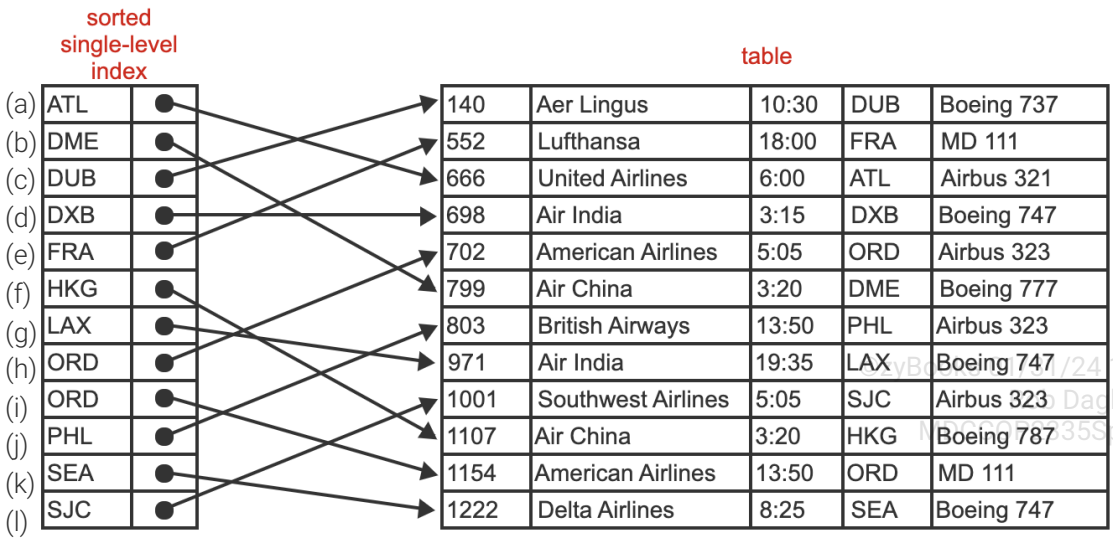
4) A table update may cause an index block split.

- True
- False

CHALLENGE ACTIVITY 38.3.1: Single-level indexes.

539740.3879454.qx3zqy7

Start



Select the index entry that refers to the row for American Airlines flight 702.

Pick

1 2 3 4 5 6

Check Next

38.4 Multi-level indexes

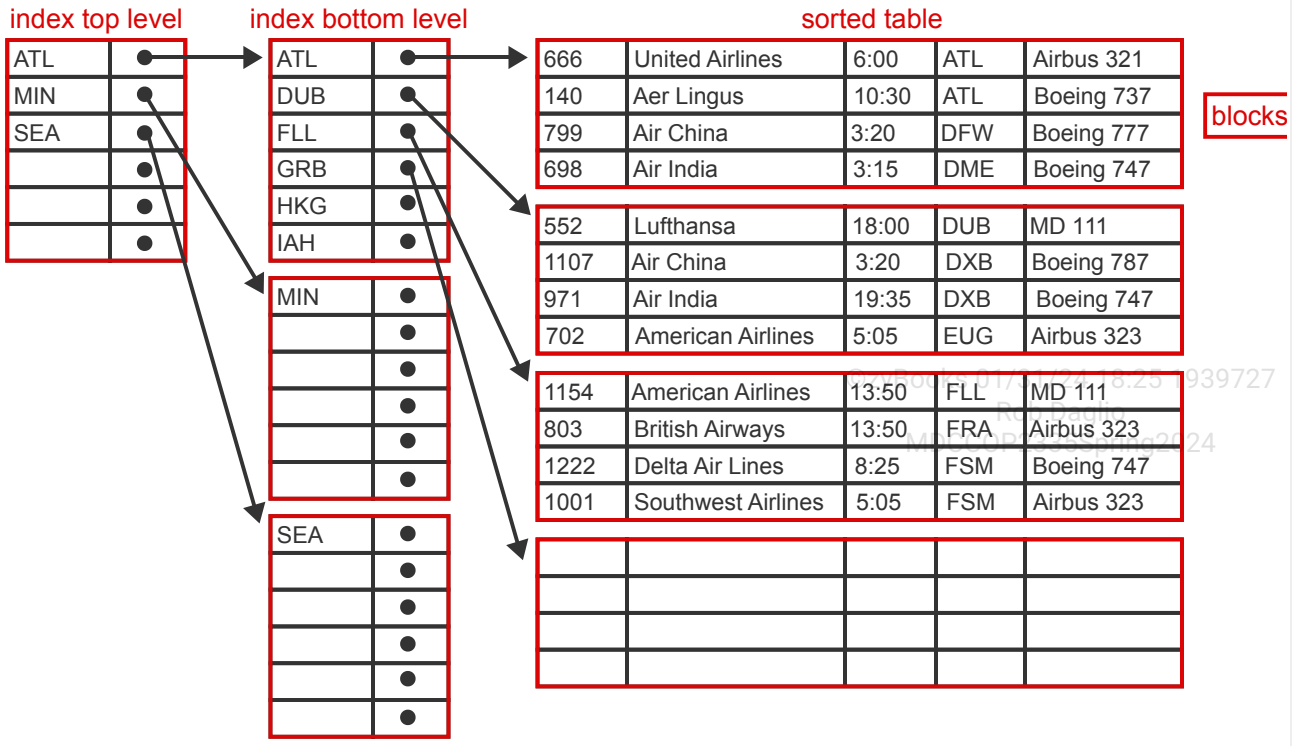
Multi-level indexes

A ***multi-level index*** stores column values and row pointers in a hierarchy. The bottom level of the hierarchy is a sorted single-level index. The bottom level is sparse for primary indexes, or dense for secondary indexes.

Each level above the bottom is a sparse sorted index to the level below. Since all levels above the bottom are sparse, levels rapidly become smaller. The top level always fits in one block.

To locate a row containing an indexed value, the database first reads the top-level block. The database compares the indexed value to entries in the block and locates the next level block containing the value. Continuing in this manner, the database eventually locates the bottom-level block containing the value. The bottom-level block contains a pointer to the correct table block.

PARTICIPATION ACTIVITY 38.4.1: Multi-level indexes.



Animation content:

Static figure:
A sorted table has four 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 airport code.

An index has entries containing an airport code and a pointer. The index has two levels. Entries in both levels are sorted by airport code. Bottom-level entries point to the table block containing the entry's airport code. Top-level entries point to the bottom-level block containing the entry's airport code.

Both index levels are sparse - entries point to index or table blocks rather than index entries or table rows.

Step 1: Flight table is sorted on non-unique column DepartureAirport. The sorted table appears. Airport codes are highlighted.

Step 2: The bottom level is a sparse index on DepartureAirport. If the table is not sorted on the index column, the bottom level must be dense. The bottom index level appears. Each entry is highlighted, along with the first row of the table block that the entry points to.

Step 3: The next higher level is a sparse index to the lower level. The top index level appears. Each entry is highlighted, along with the first entry of the bottom-level block that the entry points to.

Animation captions:

1. Flight table is sorted on non-unique column DepartureAirport.
2. The bottom level is a sparse index on DepartureAirport. If the table is not sorted on the index column, the bottom level must be dense.
3. The next higher level is a sparse index to the lower level.

PARTICIPATION ACTIVITY 38.4.2: Multi-level indexes.

1) The bottom level of a multi-level index is always sparse.
   - ○ True
   - ○ False

2) Levels above the bottom of a multi-level index are always sparse.

- True
- False

3) Each column value appears at most once in a multi-level index.

- True
- False

4) In a sparse multi-level index, each table block pointer appears exactly once.

- True
- False

Number of levels

A dense index has more bottom-level entries than a sparse index, and may have more levels. Assuming a table with 10 million rows and 400 index entries per block, a dense index has three levels:

- Level 3 is dense and has 25,000 blocks = 10 million rows / 400 index entries per block
- Level 2 is sparse and has 63 blocks = 25,000 level 3 blocks / 400 index entries per block
- Level 1 is sparse and has one block containing 63 index entries.

The number of index entries per block is called the ***fan-out*** of a multi-level index. The number of levels in a multi-level index can be computed from fan-out, number of rows, and rows per block:

- For a dense index, number of levels = $\log_{\text{fan-out}}$ (number of rows)
- For a sparse index, number of levels = $\log_{\text{fan-out}}$ (number of rows / rows per block)

In both cases, log is a fractional number and must be rounded up to the nearest integer. Both formulas assume minimal free space in the index.

Dense indexes usually have four levels or less. Sparse indexes usually have three levels or less.

Table 38.4.1: Number of levels.

dense index		rows	
		1 million	1 billion
index entries	100	3	5

per block	400	3	4

sparse index		rows	
		1 million	1 billion
index entries per block	100	3	4
	400	2	3

PARTICIPATION ACTIVITY 38.4.3: Number of levels.

Assume a table has 1,000,000 rows. Index entries are 26 bytes, and index blocks are 8 kilobytes. Use a calculator to compute logs.

1) What is the fan-out for a multi-level index?
   - Approximately 200
   - Approximately 300
   - Approximately 400

2) How many levels does a dense multi-level index have?
   - 2
   - 3
   - 4

3) If table blocks are 2 kilobytes and rows are 100 bytes, how many levels does a sparse multi-level index have?
   - 2
   - 3
   - 4

Query processing

Multi-level indexes are faster than single-level indexes on most queries. Consider the following scenario:

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

The table contains 250,000 blocks = 10 million rows × 100 bytes per row / 4,000 bytes per block.

A dense, single-level index contains 25,000 blocks = 10 million entries × 10 bytes per entry / 4,000 bytes per block.

A dense, multi-level index contains 3 levels = $\log_{\text{400 entries per index block}}$ 10,000,000 rows, rounded up.

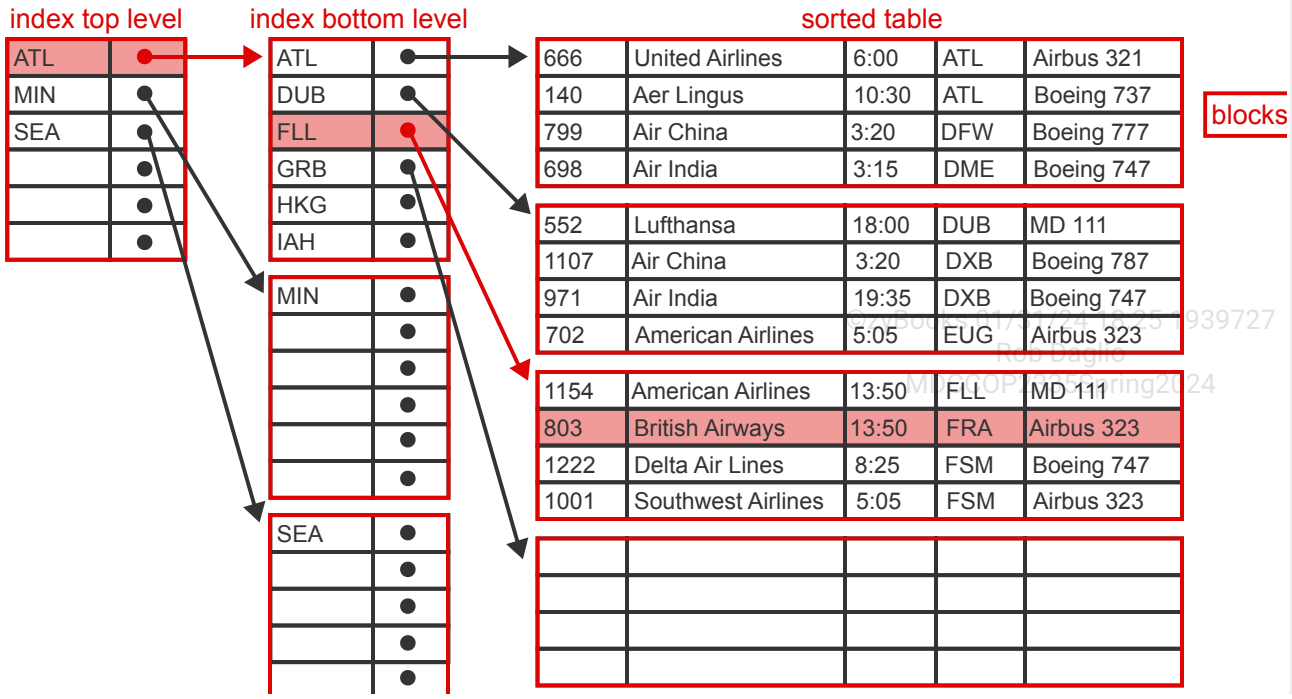
A query searches for rows containing a specific value of an indexed column. Assuming query hit ratio is low:

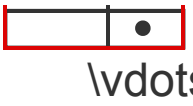
- A table scan reads at most 250,000 table blocks.
- A single-level index scan reads at most 25,000 index blocks plus a few table blocks.
- A binary search of a sorted, single-level index reads at most 15 index blocks (= $\log_2$ 25,000, rounded up) plus a few table blocks.
- A multi-level index search reads 3 index blocks plus a few table blocks.

The multi-level index search reads one index block per level. Usually the top two levels are small and retained in memory. Since the index has three levels, the query reads just one index block from storage media.

Because multi-level indexes are faster than single-level indexes on most queries, databases commonly use multi-level rather than single-level indexes.

PARTICIPATION ACTIVITY 38.4.4: Query processing with a multi-level index.





\vdots

```
SELECT AirlineName
FROM Flight
WHERE DepartureAirport = 'FRA';
```


Animation content:

Static figure:
A sorted table has four 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 airport code.

An index has entries containing an airport code and a pointer. The index has two levels. Entries in both levels are sorted by airport code. Bottom-level entries point to the table block containing the entry's airport code. Top-level entries point to the bottom-level block containing the entry's airport code.

Both index levels are sparse - entries point to index or table blocks rather than index entries or table rows.

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

Step 1: A query searches for rows containing 'FRA'. The WHERE clause is highlighted.

Step 2: The database reads the index's top level. 'FRA' is between 'ATL' and 'MIN'. FRA appears between top-level entries for ATL and MIN. The top-level ATL entry is highlighted.

Step 3: The database follows the 'ATL' pointer and reads the bottom-level block. 'FRA' is between 'FLL' and 'GRB'. The top-level entry for ATL points to a bottom-level block containing entries for FLL and GRM. FRA appears between the FLL and GRM entries. The bottom-level FLL entry is highlighted.

Step 4: The database follows the 'FLL' pointer, reads the table block, and locates the row containing 'FRA'. FRA appears next to the table block that the bottom-level index block containing FLL points to. The row in this table block containing FRA is highlighted.

Animation captions:

1. A query searches for rows containing 'FRA'.

2. The database reads the index's top level. 'FRA' is between 'ATL' and 'MIN'.
3. The database follows the 'ATL' pointer and reads the bottom-level block. 'FRA' is between 'FLL' and 'GRB'.
4. The database follows the 'FLL' pointer, reads the table block, and locates the row containing 'FRA'.

PARTICIPATION ACTIVITY 38.4.5: Query processing.

Match the search type to the maximum number of blocks read.

If unable to drag and drop, refresh the page.

Multi-level sparse index search | **Table scan** | **Single-level index binary search**

Multi-level dense index search | **Single-level index scan**

	Number of rows / rows per table block
	Number of index blocks plus referenced table blocks.
	log base 2 (number of index blocks) plus referenced table blocks.
	log base fan-out (number of rows), rounded up, plus referenced table blocks.
	log base fan-out (number of rows / rows per table block), rounded up, plus referenced table blocks.

Reset

Balanced indexes

Each path from the top-level block to a bottom-level block is called a ***branch***. Multi-level indexes are called ***balanced*** when all branches are the same length and ***imbalanced*** when branches are different lengths.

Imbalanced indexes are undesirable, since processing time is unpredictable. If a query follows a long branch, the query is relatively slow. For this reason, inserts are managed to maintain balanced indexes:

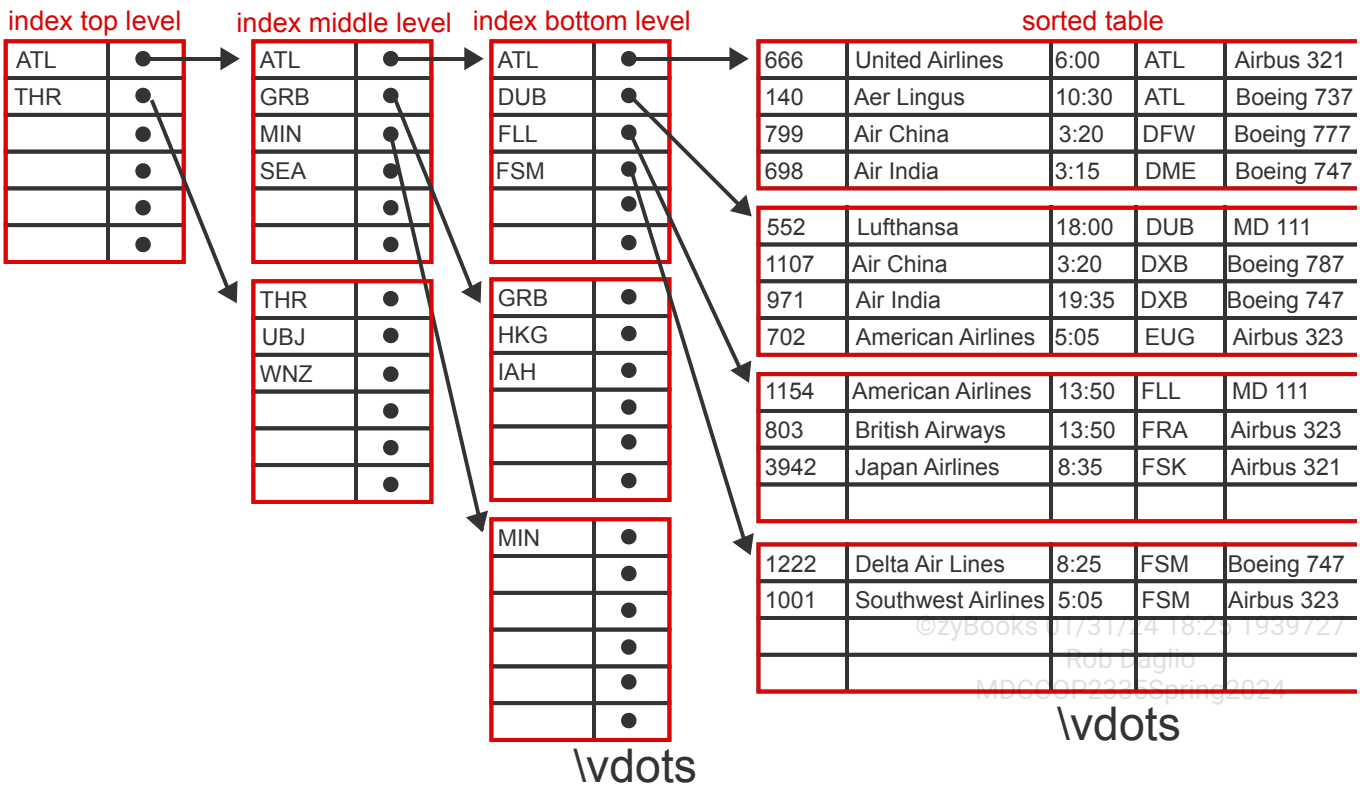
1. In a dense index, inserts always generate new bottom-level index entries. In a sparse index, inserts generate new bottom-level index entries when table blocks split.
2. If the new index entry goes in a full index block, the block splits. Half of the rows move to the new block, creating space for the entry.
3. The new block in the bottom level generates a new index entry the next level up. If the block in the next level up is full, the block splits and the process repeats.
4. If blocks are full at all index levels, the split propagates to the top level. In this case, the top-level block splits and a new level is created.

New levels are always added at the top of the hierarchy rather than the bottom of one branch. As a result, all branches are always the same length, and the index is always balanced.

Deletes may cause block mergers. Block mergers are the reverse of block splits and potentially eliminate the top level of the index. Consequently, deletes also maintain a balanced index.

Updates to an indexed column behave like a delete of the initial value followed by an insert of a new value. Since updates are implemented as deletes and inserts, updates also leave the index balanced.

PARTICIPATION ACTIVITY 38.4.6: Balancing a sparse index when inserting.




```
INSERT INTO Flight
VALUES (3942, 'Japan Airlines', '8:35', 'FSK', 'Airbus 321')
```

Animation content:

Static figure:
A sorted table has four 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 airport code.

An index has entries containing an airport code and a pointer. The index has three levels. Entries in all levels are sorted by airport code. Bottom-level entries point to the table block containing the entry's airport code. Middle-level entries point to the bottom-level block containing the entry's airport code. Top-level entries point to the middle-level block containing the entry's airport code.

All index levels are sparse - entries point to index or table blocks rather than index entries or table rows.

An SQL statement appears.
Begin SQL code:
INSERT INTO Flight
VALUES (3942, 'Japan Airlines', 8:35, 'FSK', 'Airbus 321');
End SQL code.

Step 1: The Flight table is sorted on DepartureAirport. A row is inserted with DepartureAirport 'FSK'. The index, table, and statement appear. The index has two levels, labeled top level and bottom level.

Step 2: The table block is full and splits to create space for the new row. Table block 2 splits into two blocks. The last two rows of block 2 move to the new block 3. A new row is inserted to the new space created in table block 2:
3942, Japan Airlines, 8:35, FSK, Airbus 321

Step 3: The new table block generates a new entry in the index's bottom level. The airport code FSM, from the first row of the new block 3, appears between the FLL and GRB entries in the bottom index level.

Step 4: The bottom-level index block is full and splits. Bottom-level index block 0 splits into two blocks. The last three entries of index block 0 move to the new index block 1. A new entry for FSM is inserted to the new space created in index block 0. The new entry points to the new table block 2, containing the new row.

Step 5: The new bottom-level block generates a new entry for the index's top level. Airport code GRB, from the first entry of the new index block 1, appears between entries for ATL and MIN of the index top level.

Step 6: The top-level index is full and splits. The index top level is renamed index middle level. A new level appears with caption index top level. Middle-level block 0 splits into two blocks. The last three entries of middle-level block 0 move to the new middle-level block 1. A new GRB entry is inserted to

middle-level block 0, pointing to the corresponding bottom-level block. The new top-level index has one block with entries for ATL and THR, pointing to the corresponding middle-level blocks.

Animation captions:

1. The Flight table is sorted on DepartureAirport. A row is inserted with DepartureAirport 'FSK'.
2. The table block is full and splits to create space for the new row.
3. The new table block generates a new entry in the index's bottom level.
4. The bottom-level index block is full and splits.
5. The new bottom-level block generates a new entry for the index's top level.
6. The top-level index is full and splits. A new top level is created containing entries for two blocks in the middle level.

PARTICIPATION ACTIVITY 38.4.7: Balanced indexes.

A table has a multi-level index with no free space in any index blocks.

1) If the index is sparse, an insert to the table always generates a new index level.
   - True
   - False

2) If the index is dense, an insert to the table always generates a new index level.
   - True
   - False

3) If the index is dense, an update to the indexed column always generates a new index level.
   - True
   - False

B-tree and B+tree indexes

The balanced multi-level index described above is called a B+tree. B+tree structure is derived from an earlier approach called a B-tree. The two differ as follows:

- ***B+tree***. All indexed values appear in the bottom level. Pointers to table blocks appear only in the bottom level. Since some indexed values also appear in higher levels, values are occasionally

repeated in the index.

- ***B-tree***. If an indexed value appears in a higher level, the value is not repeated at lower levels. Instead, a pointer to the corresponding table block appears in the higher level along with the value.

B-trees are more compact than B+trees since index values are not repeated. However, B+trees are simpler, since all pointers to table blocks appear in the same (bottom) level. The B+tree structure has two benefits:

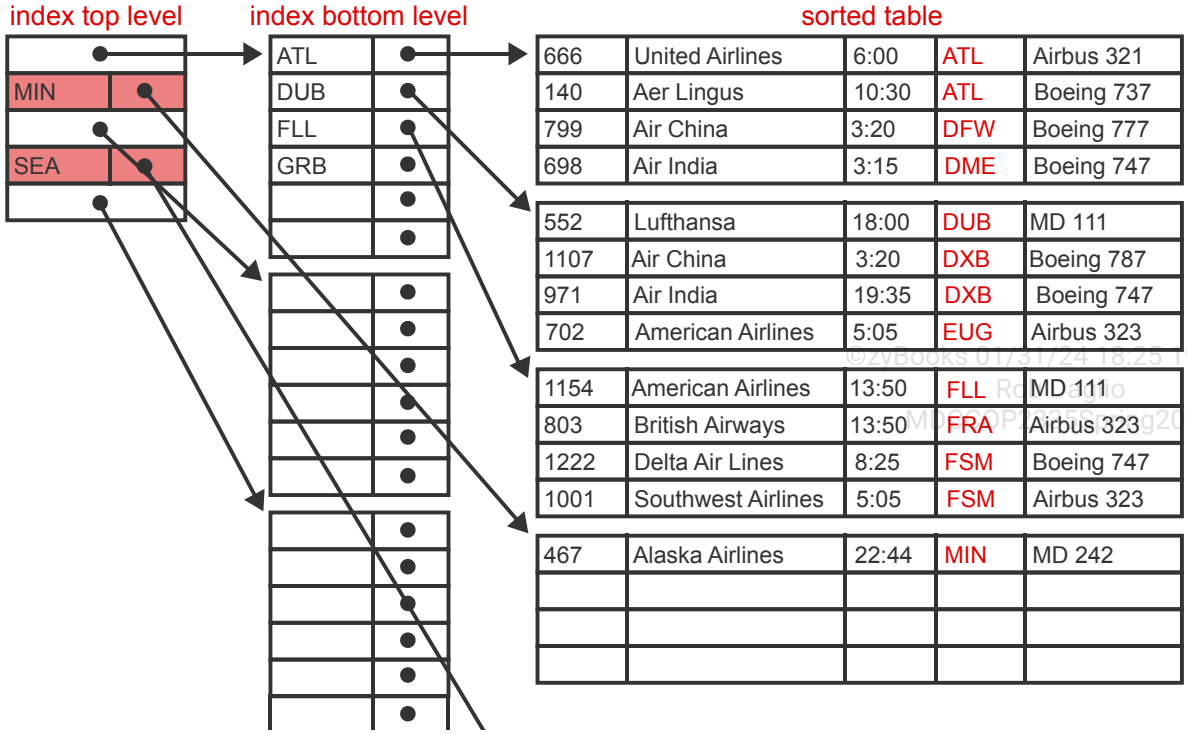
- The bottom level of a B+tree is a single-level index and can be scanned or searched.
- In a B-tree, inserts, updates, and deletes may cause a table pointer to change levels, which is hard to implement. B+trees do not have this problem, since table pointers are always in the bottom level.

Because of the advantages above, multi-level indexes are usually implemented as B+trees.

Terminology

*Although most multi-level indexes are implemented as B+trees, the term **B-tree** is commonly used and often refers to a B+tree structure. **B+tree** is commonly written as B+-tree or B^+-tree.*

PARTICIPATION ACTIVITY 38.4.8: B-tree index.



666	United Airlines	6:00	ATL	Airbus 321
140	Aer Lingus	10:30	ATL	Boeing 737
799	Air China	3:20	DFW	Boeing 777
698	Air India	3:15	DME	Boeing 747
552	Lufthansa	18:00	DUB	MD 111
1107	Air China	3:20	DXB	Boeing 787
971	Air India	19:35	DXB	Boeing 747
702	American Airlines	5:05	EUG	Airbus 323
1154	American Airlines	13:50	FLL	MD 111
803	British Airways	13:50	FRA	Airbus 323
1222	Delta Air Lines	8:25	FSM	Boeing 747
1001	Southwest Airlines	5:05	FSM	Airbus 323
467	Alaska Airlines	22:44	MIN	MD 242

Animation content:

Static figure:
A sorted table has four 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 airport code.

An index has entries containing an airport code and a pointer. The index has two levels. Entries in both levels are sorted by airport code. Bottom-level entries point to the table block containing the entry's airport code. The bottom level is identical to the bottom level in prior animations in this section.

The top level is different from prior animations in this section. The top level has five entries. Entries 0, 2, and 4 contain a pointer only, pointing to bottom-level index blocks 0, 1, and 2, respectively. Entries 1 and 3 contain an airport code and a pointer. These entries point to the table block that begins with the corresponding airport code.

Both index levels are sparse - entries point to index or table blocks rather than index entries or table rows.

Step 1: The bottom level of a B-tree is like the bottom level of a B+tree. The table and index bottom level appear.

Step 2: Higher levels of a B-tree alternate index block pointers and table block pointers. The index top level appears. Each top-level entry is highlighted, along with the table or index block that the entry points to.

Step 3: Entries that appear in higher levels are not repeated in the bottom level. Top-level entries 1 and 3, containing an airport code and a pointer to a table block, are highlighted. These entries point directly to the corresponding table block, and do not appear in the index bottom level.

Animation captions:

1. The bottom level of a B-tree is like the bottom level of a B+tree.
2. Higher levels of a B-tree alternate index block pointers and table block pointers.
3. Entries that appear in higher levels are not repeated in the bottom level.

PARTICIPATION ACTIVITY 38.4.9: B-tree and B+tree indexes.

1) Which type of index can have table block pointers at any level?

- B-tree
- B+tree
- Both B-tree and B+tree

2) In informal use, the term 'B-tree' commonly means:

- B-tree index
- B+tree index
- Either a B-tree or a B+tree index

3) Which index structure enables an index scan using the bottom level only?

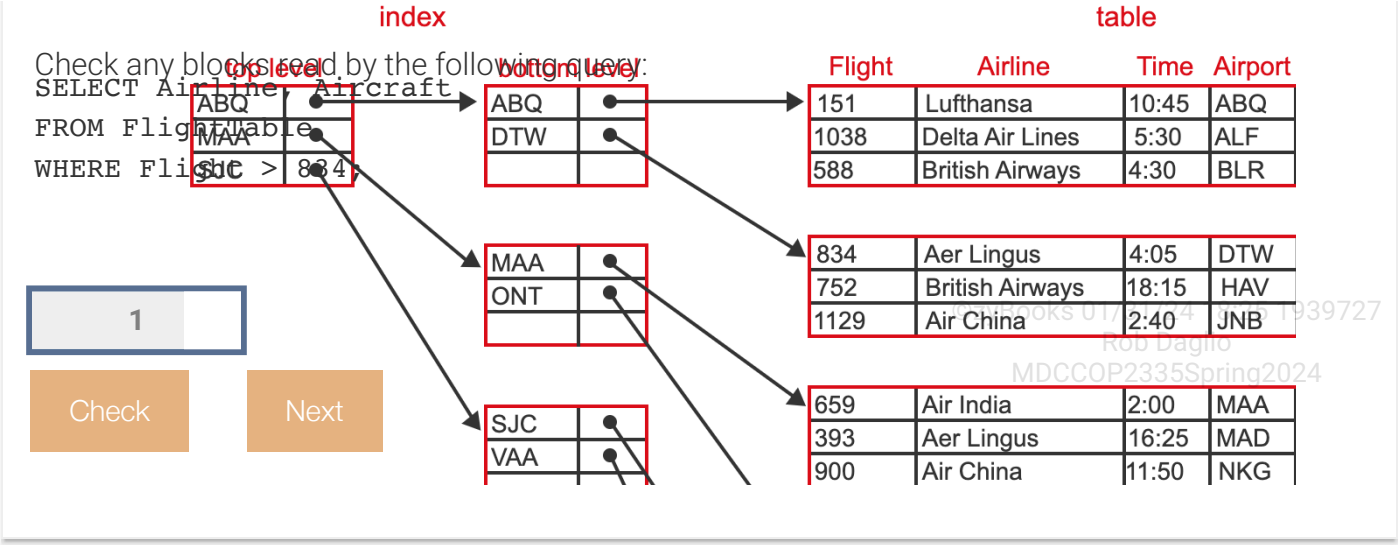
- B-tree
- B+tree
- Both B-tree and B+tree

CHALLENGE ACTIVITY 38.4.1: Multi-level indexes.

539740.3879454.qx3zqy7

Start

The table is sorted on Airport.
The index on Airport is sparse.



38.5 Other indexes

Hash indexes

The multi-level index is the most commonly used index type. Several additional index types are used less often but supported by many databases:

- Hash index
- Bitmap index
- Logical index
- Function index

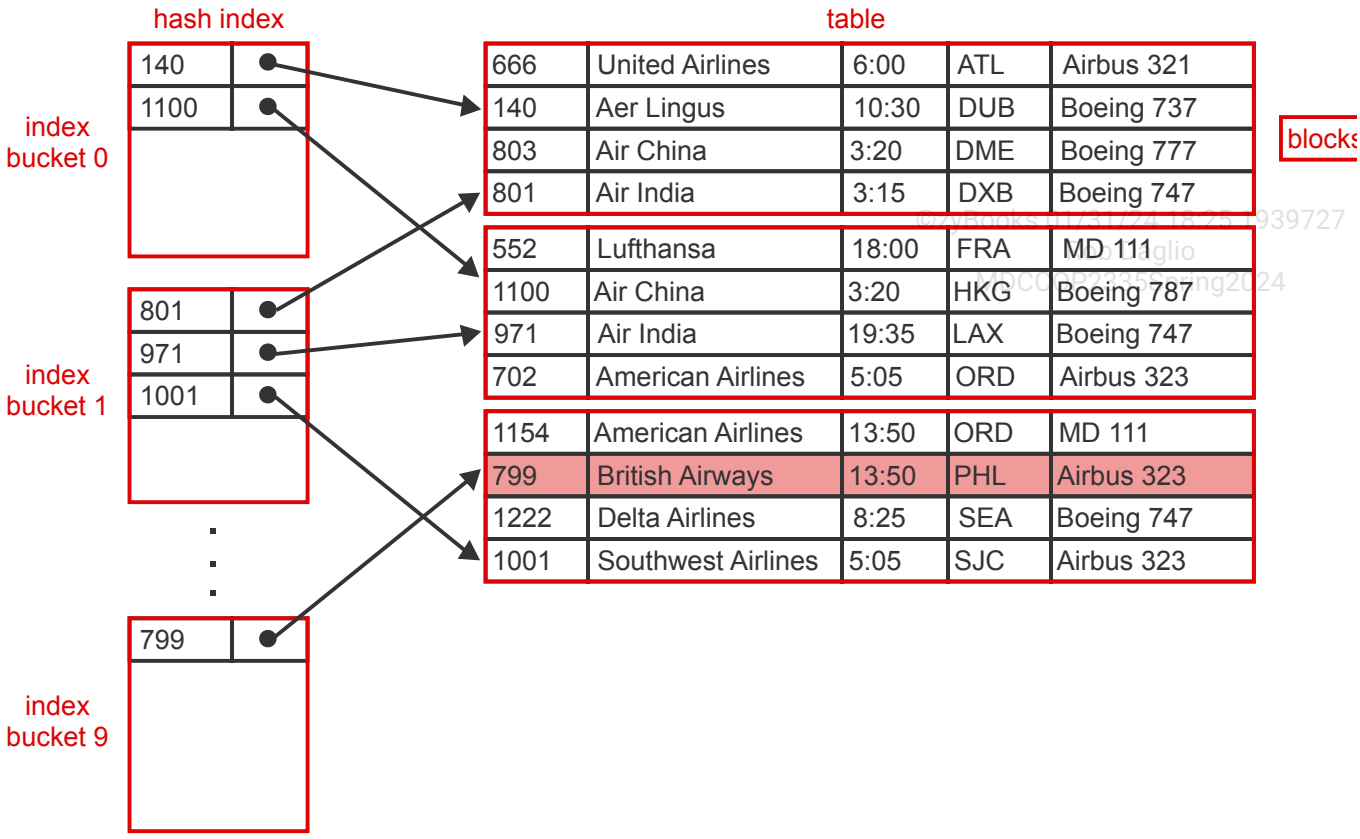
In a ***hash index***, index entries are assigned to buckets. A ***bucket*** is a block or group of blocks containing index entries. Initially, each bucket has one block. As an index grows, some buckets eventually fill up, and additional blocks are allocated and linked to the initial block.

The bucket containing each index entry is determined by a ***hash function***, which computes a bucket number from the value of the indexed column. To locate a row containing a column value, the database:

1. Applies the hash function to the column value to compute a bucket number.
2. Reads the index blocks for the bucket number.
3. Finds the index entry for the column value and reads the table block pointer.
4. Reads the table block containing the row.

A hash index is similar to a hash table, described in another section. However, a hash index stores *index entries* in each bucket, while a hash table stores *table rows* in each bucket.

PARTICIPATION ACTIVITY 38.5.1: Hash index.



Animation content:

Static figure:
A 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 airport code.

A hash index appears with buckets 0, 1, and 9. Buckets 2 through 8 are not shown. Each bucket has entries with a flight number and pointer to the table block containing the corresponding row. The last digits of all flight numbers in each hash bucket are the same as the bucket number.

Step 1: The table is clustered on DepartureAirportCode. The table appears. Airport codes are highlighted.

Step 2: The hash index is on FlightNumber. The hash function is modulo 10. Flight numbers are highlighted.

Step 3: Index entries for flight numbers ending in 0 go in bucket 0. Bucket 0 appears. All flight numbers in the index entries end in 0. Index entries point to table rows containing the corresponding flight number.

Step 4: Index entries for flight numbers ending in 1 go in bucket 1. Bucket 1 appears. All flight

numbers in the index entries end in 1. Index entries point to table rows containing the corresponding flight number.

Step 5: Since the hash function is modulo 10, the hash index has 10 buckets. Bucket 9 appears. All flight numbers in the index entries end in 9. Index entries point to table rows containing the corresponding flight number.

Animation captions:

1. The table is clustered on DepartureAirportCode.
2. The hash index is on FlightNumber. The hash function is modulo 10.
3. Index entries for flight numbers ending in 0 go in bucket 0.
4. Index entries for flight numbers ending in 1 go in bucket 1.
5. Since the hash function is modulo 10, the hash index has 10 buckets.

Terminology

*Sometimes the term hash index is used to mean a hash key, but the two terms are different. A **hash index** is an index that is structured using a hash function. A **hash key** is a column that determines the physical location of rows in a hash table.*

PARTICIPATION ACTIVITY 38.5.2: Hash indexes.

1) A hash table can have a hash index.
   - True
   - False

2) A primary index can be structured as a hash index.
   - True
   - False

3) A hash key can be structured as a multi-level index.
   - True
   - False

4) A hash index can be sparse.

- ○ True
- ○ False

Bitmap indexes

A ***bitmap index*** is a grid of bits:

- Each index row corresponds to a unique table row. If the table's primary key is an integer, the index row number might be the primary key value. Alternatively, the index row number might be an internal table row number, maintained by the database.
- Each index column corresponds to a distinct table value. Ex: If the index is on AirportCode, each index column corresponds to a different three-letter airport code. The mapping of index column numbers to table values is computed with a function or stored in an internal 'lookup' table.

Bitmap indexes contain ones and zeros. 'One' indicates that the table row corresponding to the index row number contains the table value corresponding to the index column number. 'Zero' indicates the row does not contain the value.

To locate rows containing a table value, the database:

1. Determines the index column corresponding to the table value.
2. Reads the index column and finds index rows that are set to 'one'.
3. Determines table rows corresponding to the index rows.
4. Determines pointers to blocks containing the table rows.
5. Reads the blocks containing the table rows.

An efficient bitmap index has two characteristics:

- The database can quickly determine the block containing a table row from the index row number (steps 3 and 4). Ex: The index row number is the hash key for a hash table. The block is determined by applying the hash function to the row number. Ex: The index row number is the table primary key. The block is determined with a primary index.
- The indexed column contains relatively few distinct values, typically tens or hundreds. If the indexed column contains thousands of distinct values, the bitmap index is large and inefficient.

Bitmap indexes with the above characteristics enable fast reads. Ex: A table with 10 million rows has a bitmap index on a column with 100 distinct values. The index contains 125 million bytes (10 million rows × 100 column values × 1 bit per index entry / 8 bits per byte) and can easily be retained in memory.

PARTICIPATION ACTIVITY 38.5.3: Bitmap index.

bitmap index

	ATL	DUB	LAX	MSN	ORD	SEA	SJC
0	1	0	0	0	0	0	0
1	0	1	0	0	0	0	0
2	0	0	1	0	0	0	0
3	0	0	0	1	0	0	0
4	0	0	0	1	0	0	0
5	1	0	0	0	0	0	0
6	0	0	0	0	0	1	0
7	0	0	0	1	0	0	0
8	0	0	0	0	1	0	0
9	1	0	0	0	0	0	0
10	0	0	1	0	0	0	0
11	0	0	0	0	0	0	1

table

blocks

0	666	United Airlines	6:00	ATL	Airbus 321
1	140	Aer Lingus	10:30	DUB	Boeing 737
2	803	Air China	3:20	LAX	Boeing 777
3	801	Air India	3:15	MSN	Boeing 747
4	552	Lufthansa	18:00	MSN	MD 111
5	1100	Air China	3:20	ATL	Boeing 787
6	971	Air India	19:35	SEA	Boeing 747
7	702	American Airlines	5:05	MSN	Airbus 323
8	1154	American Airlines	13:50	ORD	MD 111
9	799	British Airways	13:50	ATL	Airbus 323
10	1222	Delta Airlines	8:25	LAX	Boeing 747
11	1001	Southwest Airlines	5:05	SJC	Airbus 323

Animation content:

Static figure:
A sorted table has three blocks. Each block contains four rows. Each row contains row number, flight number, airline name, departure time, airport code, and aircraft type. Rows are numbered 0 through 11 and are sorted on row number.

A bitmap index appears. The index is a grid. Each grid cell contains 0 or 1. Grid rows are numbered 0 through 11. Grid columns are labeled with airport codes that appear in the table: ATL, DUB, LAS, MSN, ORD, SEA, and SJC.

Step 1: The database maintains row numbers internally. The table appears. Row numbers are highlighted.

Step 2: A bitmap index is a grid of bits. Bitmap index rows correspond to internal table row numbers. The bitmap index appears with no values in the grid. Row numbers appear.

Step 3: Bitmap index columns correspond to table values. Grid column labels appear.

Step 4: Table row 0 contains ATL, so the corresponding index bit is 1. 1 appears in grid cell (0, ATL). Table row 0 contains airport code ATL and is highlighted.

Step 5: Table row 1 contains DUB, so the corresponding index bit is 1. 1 appears in grid cell (1, DUB). Table row 1 contains airport code DUB and is highlighted.

Step 6: Each 1 in the bitmap index indicates the corresponding value appears in the table row. For

each row in the table, a 1 appears in the grid cell for the row's number and airport code.

Step 7: Each 0 in the bitmap index indicates the corresponding value does not appear in the table row. 0 appears in the remaining grid cells.

Animation captions:

1. The database maintains row numbers internally.
2. A bitmap index is a grid of bits. Bitmap index rows correspond to internal table row numbers.
3. Bitmap index columns correspond to table values.
4. Table row 0 contains ATL, so the corresponding index bit is 1.
5. Table row 1 contains DUB, so the corresponding index bit is 1.
6. Each 1 in the bitmap index indicates the corresponding value appears in the table row.
7. Each 0 in the bitmap index indicates the corresponding value does not appear in the table row.

PARTICIPATION ACTIVITY 38.5.4: Bitmap indexes.

1) Refer to the bitmap index in the above animation. The three 1's in the MSN column means:
   - The value MSN appears in three table blocks.
   - The value MSN appears in three table rows.
   - The value MSN appears in three buckets.

2) How many 1's can appear in a single row of a bitmap index?
   - Exactly one
   - Zero or one
   - One or many

3) California has 26 million licensed drivers and 2,597 ZIP codes. The California Department of Motor Vehicles tracks one address for every licensed driver in a table with a ZipCode column. How many bytes are in a bitmap index on ZipCode?

- ○ Approximately 26 million
- ○ Exactly 2,597 / 8
- ○ Approximately 8.4 billion

Logical indexes

A single- or multi-level index normally contains pointers to table blocks and is called a ***physical index***.

A ***logical index*** is a single- or multi-level index in which pointers to table blocks are replaced with primary key values. Logical indexes are always secondary indexes and require a separate primary index on the same table. To locate a row containing a column value, the database:

1. Looks up the column value in the logical index to find the primary key value.
2. Looks up the primary key value in the primary index to find the table block pointer.
3. Reads the table block containing the row.

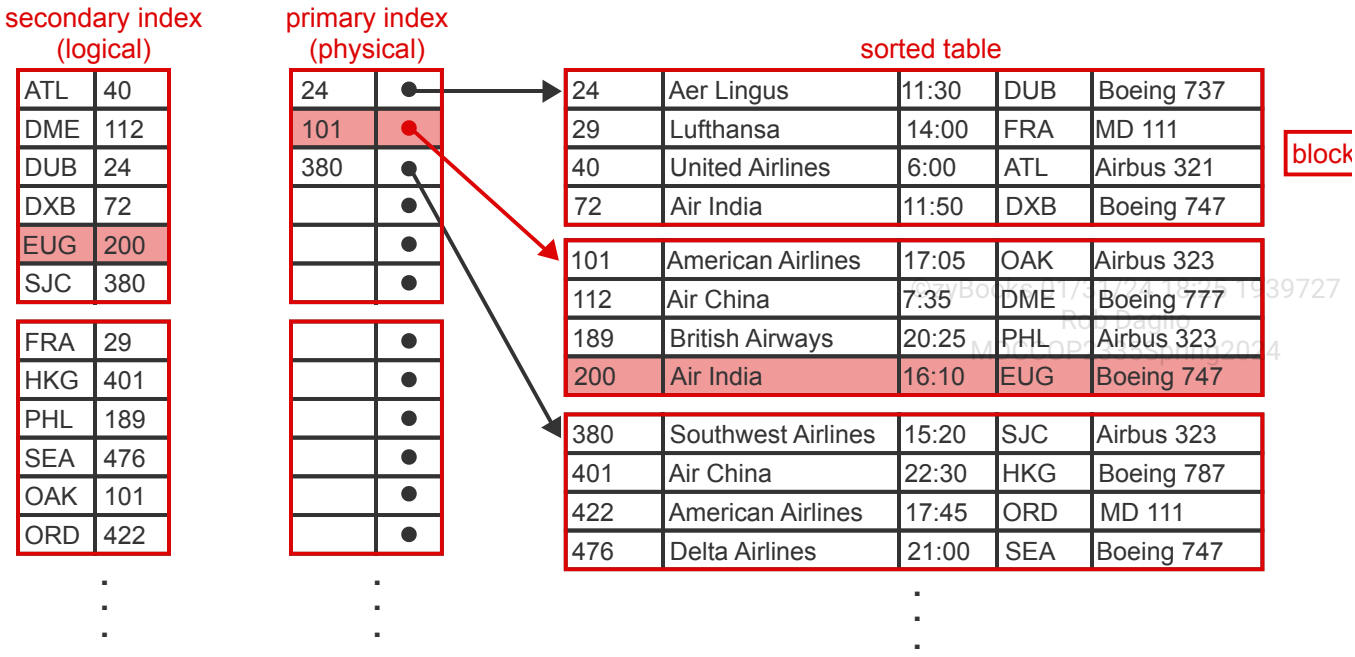
Logical indexes change only when primary key values are updated, which occurs infrequently. Physical indexes change whenever a row moves to a new block, which occurs in several ways:

- *A row is inserted into a full block*. To create space for the new row, the block splits and some rows move to a new block.
- *The sort column is updated*. When the sort column is updated, the row may move to a new block to maintain sort order.
- *The table is reorganized*. Occasionally, a database administrator may physically reorganize a table to recover deleted space or order blocks contiguously on magnetic disk.

If a table has several indexes, the time required to update physical indexes is significant, and logical indexes are more efficient.

On read queries, a logical index requires an additional read of the primary index and is slower than a physical index. However, the primary index is often retained in memory, mitigating the cost of the additional read.

PARTICIPATION ACTIVITY 38.5.5: Logical index.



Animation content:

Static figure:
A 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 table is sorted on fight number.

A sparse index appears, with caption primary index (physical). The index contains one level. Each entry contains a flight number and pointer to the table block that begins with the corresponding row.

A dense index appears, with caption secondary index (logical). The index contains one level. Each entry contains an airport code and a flight number. Every airport code in the table appears in this index, along with the flight number for the corresponding row.

Step 1: The table is sorted on FlightNumber, the primary key. This example assumes FlightNumber is unique across all airlines. The sorted table appears. Flight numbers are highlighted.

Step 2: The table has primary index on FlightNumber. The primary index is physical and sparse. The primary index appears.

Step 3: The table has secondary index on DepartureAirportCode. The secondary index is logical and dense. The secondary index appears.

Step 4: To find the row containing EUG, the database finds the primary key 200 in the secondary index, then finds the block pointer in the primary index leading to the block containing 200. In the secondary index, entry (EUG, 200) is highlighted. Next to the primary index, 200 appears between entries for 101 and 380. Primary index entry 101 is highlighted and points to the table block beginning with flight number 101. The last row of this table block contains flight number 200 and is highlighted.

Animation captions:

1. The table is sorted on FlightNumber, the primary key. This example assumes FlightNumber is unique across all airlines.
2. The table has primary index on FlightNumber. The primary index is physical and sparse.
3. The table has secondary index on DepartureAirportCode. The secondary index is logical and dense.
4. To find the row containing EUG, the database finds the primary key 200 in the secondary index, then finds the block pointer in the primary index leading to the block containing 200.

PARTICIPATION ACTIVITY 38.5.6: Logical indexes.

If unable to drag and drop, refresh the page.

Primary index **Secondary index** **Logical index** **Physical index**

	An index on a unique sort column.
	An index on a non-sort column.
	An index with primary key values rather than block pointers.
	An index with table block pointers.

Reset

Function indexes

In some cases, values specified in a WHERE clause may be in a different format or units than values stored in the column. Ex:

- The WHERE clause specifies values in upper case, but the column contains mixed upper and lower case characters.
- The WHERE clause specifies values as percentages, from 0 to 100, but the column contains values from 0 to 1.

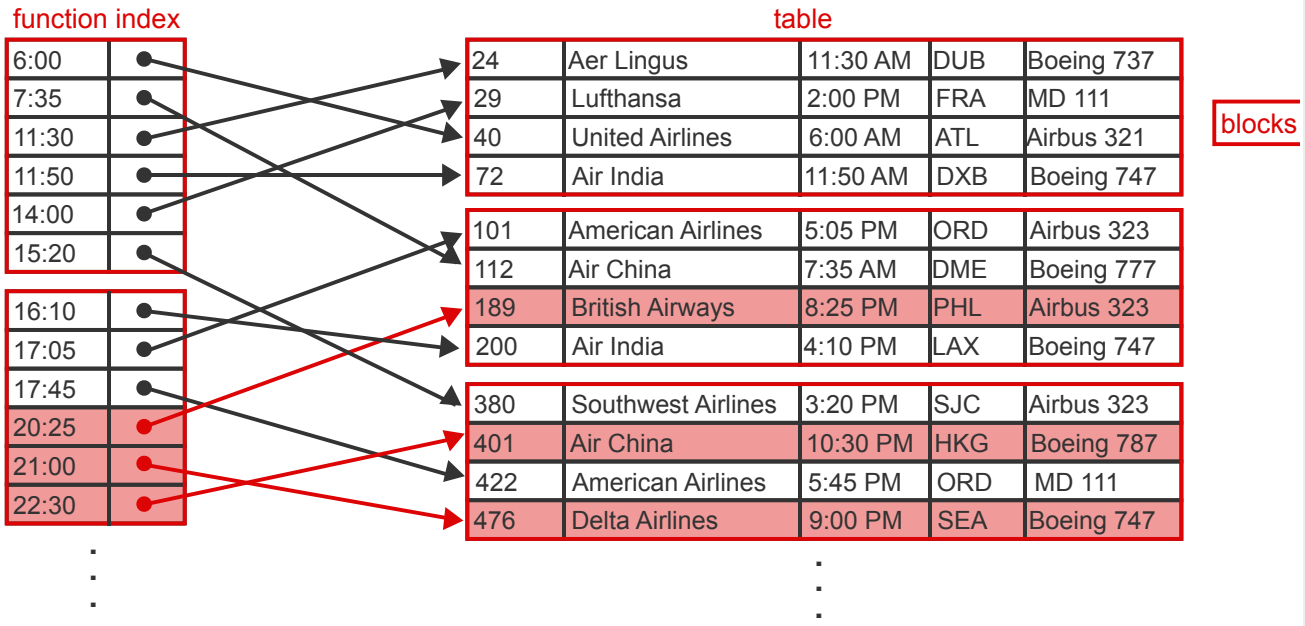
In the above examples, index entries do not match values in the WHERE clause, so the database cannot use the index to execute the query.

To address this problem, some databases support function indexes. In a ***function index***, the database designer specifies a function on the column value. Index entries contain the result of the function applied to column values, rather than the column values.

Ex: Column values are stored as decimal numbers between 0 and 1, but users specify percentages as integers between 0 and 100 in queries. The database designer specifies a function index that multiplies column values by 100 and converts the result to an integer. The index contains integers between 0 and 100, so the database can use the function index to process queries.

In principle, functions can be used with any index type, including single-level, multi-level, hash, bitmap, and logical indexes. In practice, support varies by database.

PARTICIPATION ACTIVITY 38.5.7: Function index.




```
SELECT FlightNumber, AirlineName
FROM Flight
WHERE DepartureTime > "20:00"
```

Animation content:

Static figure:
A 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 table is sorted on fight number.

A dense index appears with caption function index. Index entries contain a time and pointer to the corresponding table row. Time in the index is formatted on a 24-hour clock. Ex: 16:10. Time in the table is formatted AM/PM. Ex: 4:10 PM.

An SQL statement appears.
Begin SQL code:
SELECT FlightNumber, AirlineName
FROM Flight
WHERE DepartureTime > '20:00';
End SQL code.

Step 1: The table contains times in AM/PM format. The table appears. Time values are highlighted.

Step 2: Queries specify time in 24-hour format. The SQL statement appears. The WHERE clause is highlighted.

Step 3: The function index converts AM/PM format to 24-hour format. Each index entry, along with the corresponding table row, is highlighted.

Step 4: The function index format is consistent with the WHERE clause, so the database can use the index.

Animation captions:

1. The table contains times in AM/PM format.
2. Queries specify time in 24-hour format.
3. The function index converts AM/PM format to 24-hour format.
4. The function index format is consistent with the WHERE clause, so the database can use the index.

PARTICIPATION ACTIVITY 38.5.8: Function indexes.

1) Refer to the animation above. Which table blocks does the following query read?

```
SELECT FlightNumber, AirlineName
FROM Flight
WHERE DepartureTime > "12:00"
AND DepartureTime < "16:00";
```


- Block 0
- Block 1
- Blocks 0 and 2

2) Which index types can also be function indexes?

- Only function index types
- All index types except bitmap
- Hash, bitmap, single-level, multi-level, and logical index types

3) When should a database designer consider a function index?

- The display format is different than storage format
- The WHERE clause format is different than storage format
- The input format is different than storage format

CHALLENGE ACTIVITY 38.5.1: Other indexes.

539740.3879454.qx3zqy7

Start

Given that the hash index is on FlightNumber, and the hash function is modulo 10, select the row(s) with a hash index in bucket 8.

table

662	United Airlines	6:00	ATL	Airbus 321
548	Aer Lingus	10:30	DUB	Boeing 737
378	Air China	3:20	DME	Boeing 777
851	Air India	3:15	DXB	Boeing 747
472	Lufthansa	18:00	FRA	MD 111
211	Air China	3:20	HKG	Boeing 787
506	Air India	19:35	LAX	Boeing 747
188	American Airlines	5:05	ORD	Airbus 323
283	American Airlines	13:50	ORD	MD 111
685	British Airways	13:50	PHL	Airbus 323
596	Delta Airlines	8:25	SEA	Boeing 747
307	Southwest Airlines	5:05	SJC	Airbus 323

1 2 3 4

Check Next

38.6 Tablespaces and partitions

Tablespaces

Tablespaces and partitions are supported by most databases but are not specified in the SQL standard. Most implementations are similar, but SQL syntax and capabilities vary. This section describes the MySQL implementation.

A ***tablespace*** is a database object that maps one or more tables to a single file. The CREATE TABLESPACE statement names a tablespace and assigns the tablespace to a file. The CREATE TABLE statement assigns a table to a tablespace. Indexes are stored in the same tablespace as the indexed table.

Figure 38.6.1: SQL for tablespaces.

```
CREATE TABLESPACE TablespaceName
[ ADD DATAFILE 'FileName' ];

CREATE TABLE TableName
( ColumnName ColumnDefintion, ...
)
[ TABLESPACE TablespaceName ];
```

By default, most databases automatically create one tablespace for each table, so each table is stored in a separate file. Database administrators can manually create tablespaces and assign one or multiple tables to each tablespace. Database administrators can improve query performance by assigning frequently accessed tables to tablespaces stored on fast storage media.

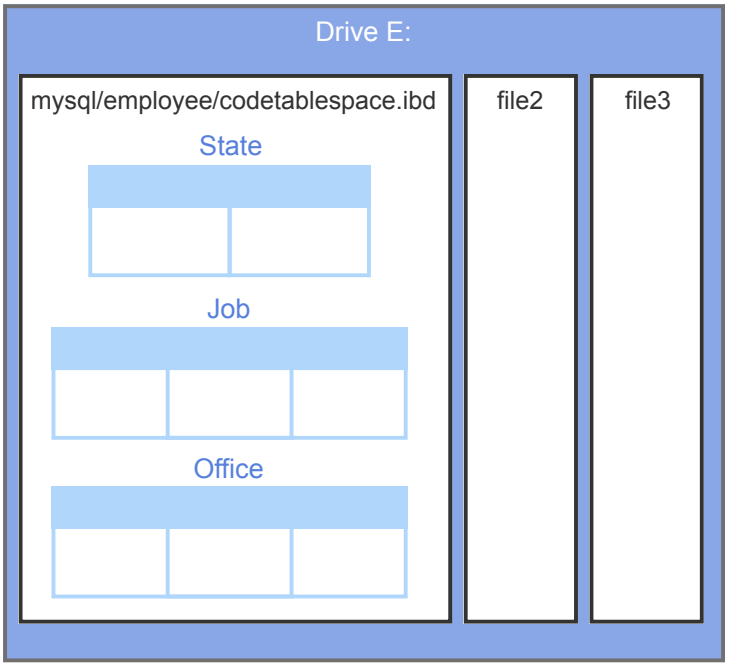
In most cases, databases perform better with a single table per tablespace:

- Individual tables can be backed up independently of other tables.
- When a table is dropped, the associated file is deleted and storage is released. When multiple tables are stored in one tablespace, all tables must be dropped to release storage.

- Concurrent updates of multiple tables are usually faster when each table is stored in a separate file.
- Blocks of a new file are usually allocated contiguously on a few tracks of a disk drive. As files are updated, blocks become scattered, or ***fragmented***, across many tracks. Queries that scan tables on heavily fragmented files are slow because the disk drive must read many tracks. When tables are updated, storing one table per file minimizes fragmentation and optimizes table scans.

In some cases, assigning multiple tables to one tablespace can improve performance. Each tablespace must be managed by the database and incurs a small amount of overhead. Storing many small tables in one tablespace reduces overhead and, if the tables are commonly accessed in the same query, may improve query performance. If the tables are read-only, assigning the tables to one tablespace does not increase fragmentation.

PARTICIPATION ACTIVITY 38.6.1: Assigning three tables to the same tablespace.




```
CREATE TABLESPACE CodeTablespace
ADD DATAFILE

'E:/mysql/employee/codetablespace.ibd'
CREATE TABLE State (
   StateCode CHAR(2),
   StateName VARCHAR(20)
)
TABLESPACE CodeTablespace;


CREATE TABLE Job (
   JobCode SMALLINT,
   JobTitle VARCHAR(30),
   JObLevel SMALLINT
)
TABLESPACE CodeTablespace;

CREATE TABLE Office (
   OfficeCode CHAR(8),
   OfficeName VARCHAR(20),
   OfficeType VARCHAR(4)
)
TABLESPACE CodeTablespace;
```

Animation content:

Static figure:
A disk drive image appears, with caption Drive E:. Inside the disk drive, three files appear, with captions mysql/employee/codetablespace.ibd, file2, and file3. Tables State, Job, and Office appear inside file mysql/employee/codetablespace.ibd.

A CREATE TABLESPACE statement appears:

Begin SQL code:
```
CREATE TABLESPACE CodeTablespace
ADD DATAFILE
  'E:/mysql/employee/codetablespace.ibd'
```
End SQL code.

Three CREATE TABLE statements appear:
Begin SQL code:
```
CREATE TABLE State (
   StateCode CHAR(2),
   StateName VARCHAR(20)
)
TABLESPACE CodeTablespace;

CREATE TABLE Job (
   JobCode SMALLINT,
   JobTitle VARCHAR(30),
   JObLevel SMALLINT
)
TABLESPACE CodeTablespace;

CREATE TABLE Office (
   OfficeCode CHAR(8),
   OfficeName VARCHAR(20),
   OfficeType VARCHAR(4)
)
TABLESPACE CodeTablespace;
```
End SQL code.

Step 1: CodeTablespace contains small, read-only code tables and is stored on a flash drive for fast reads. Drive E, file mysql/employee/codetablespace.ibd, and the CREATE TABLESPACE statement appear.

Step 2: The State table is assigned to CodeTablespace. The first CREATE TABLE statement appears. The State table appears in file mysql/employee/codetablespace.ibd.

Step 3: Additional tables are assigned to the same tablespace to reduce overhead. The second and third CREATE TABLE statements appear. The Job and Office tables appear in file mysql/employee/codetablespace.ibd.

Animation captions:

1. CodeTablespace contains small, read-only code tables and is stored on a flash drive for fast reads.
2. The State table is assigned to CodeTablespace.

3. Additional tables are assigned to the same tablespace to reduce overhead.

PARTICIPATION ACTIVITY 38.6.2: Tablespaces.

1) How many tables can be stored in one tablespace?
   - At least zero, at most one
   - At least zero, at most many
   - At least one, at most many

2) The Airport table has two thousand rows. The Airline table has ten thousand rows. Both tables are updated infrequently. Should Airport and Airline be stored in the same tablespace?
   - Always
   - Sometimes
   - Never

3) The Booking table has ten million rows and is updated frequently. The Airline table has ten thousand rows and is updated infrequently. Should Booking and Airline be stored in the same tablespace?
   - Always
   - Sometimes
   - Never

Partitions

A ***partition*** is a subset of table data. One table has many partitions that do not overlap and, together, contain all table data. A ***horizontal partition*** is a subset of table rows. A ***vertical partition*** is a subset of table columns. MySQL and most relational databases partition tables horizontally, not vertically.

Each partition is stored in a separate tablespace, specified either explicitly by the database administrator or automatically by the database. When a table is partitioned, table indexes are also partitioned. Each partition contains index entries only for rows in the partition.

Partitions can be defined in several ways. Often, rows are assigned to partitions based on values of a specific column. Each partition may be associated with a continuous range of values or an explicit list of

values. Ex:

- The Employee table has a HireDate column. One partition is created for each year. Employees are assigned to partitions based on the year hired.
- The Employee table has an OfficeID column with 10 possible values. One partition is created for each office. Employees working in the same office are stored in the same partition.

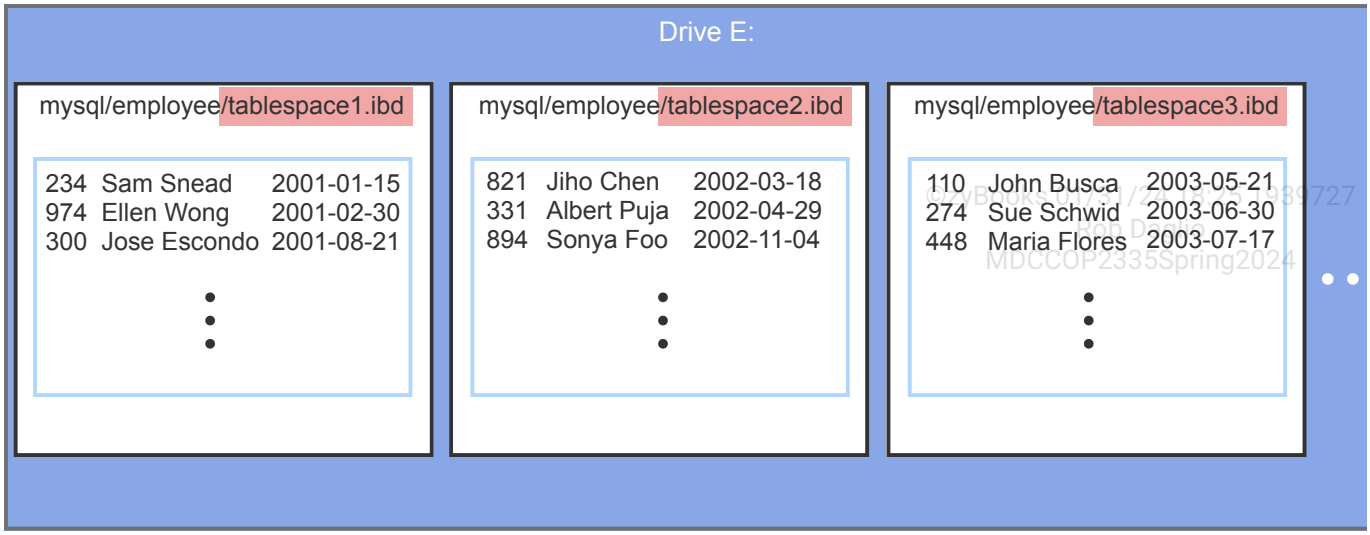
Partitions improve query performance by reducing the amount of data accessed by INSERT, UPDATE, DELETE, and SELECT statements. Ex: The Sales table contains sales transactions for the past ten years, with one partition for each year. Most queries access current year sales only. The current-year partition is a tenth of the table size, so current-year queries access less data and execute faster.

Terminology

*The term **partition** means either an individual subset of a table or, collectively, all subsets of a table. Usually, the meaning is clear from context. In this material, partition means an individual subset.*

*A shard is similar to a partition. Like a partition, a **shard** is a subset of table data, usually a subset of rows rather than columns. Unlike partitions, which are stored on different storage devices of a single computer, shards are stored on different computers of a distributed database.*

PARTICIPATION ACTIVITY 38.6.3: Employee table partitioned by hire date.



Animation content:

Static figure:
A disk drive image appears, with caption Drive E:. Inside the disk drive, three files appear, with captions mysql/employee/codetablespace1.ibd, mysql/employee/codetablespace2.ibd, and mysql/employee/codetablespace3.ibd.

Each file has rows with employee number, employee name, and a date. All dates in the first file are in 2001. All dates in the second file are in 2002. All dates in the third file are in 2003.

Step 1: The Employee table is partitioned on hire date. Drive E appears with table rows. The files do not appear.

Step 2: Each partition corresponds to one year. The year value in each row is highlighted.

Step 3: Each partition is stored in a separate tablespace. Files appear around the rows. The last part of each file name, tablespaceN, is highlighted.

Animation captions:

1. The Employee table is partitioned on hire date.
2. Each partition corresponds to one year.
3. Each partition is stored in a separate tablespace.

PARTICIPATION ACTIVITY 38.6.4: Partitions.

1) How many partitions can one table have?
   - At least zero, at most many
   - At least one, at most many
   - Always many

2) The Airport table has two thousand rows and is updated infrequently. Should Airport be partitioned?
   - Always
   - Sometimes
   - Never

3) The Reservation table has one hundred million rows and is updated frequently. Most queries access recent bookings. Should Reservation be partitioned?

- Yes, Reservation should be partitioned on the ReservationDate column
- Yes, Reservation should be partitioned on the FlightNumber column
- No

Partition types

To partition a table, the database administrator specifies a ***partition expression*** based on one or more ***partition columns***. The partition expression may be simple, such as the value of a single partition column, or a complex expression based on several partition columns. Rows are assigned to partitions in one of the following ways:

- A ***range partition*** associates each partition with a range of partition expression values. The VALUES LESS THAN keywords specify the upper bound of each range. The MAXVALUE keyword represents the highest column value, and VALUES LESS THAN MAXVALUE specifies the highest range. Each partition is explicitly named by the database administrator.
- A ***list partition*** associates each partition with an explicit list of partition expression values using the VALUES IN keywords. Like a range partition, each partition is explicitly named.
- A ***hash partition*** requires a partition expression with positive integer values. The database administrator specifies the number of partitions, N, and partitions are automatically named p0 through p(*N*-1). The partition number for each row is computed as: (partition expression value) modulo N.

Range, list, and hash partitions are supported in most relational databases. The range partition is commonly used, often with a simple partition expression based on a date column. Ex: Bank transactions are partitioned on transaction date. Each partition contains all transactions for a calendar month.

MySQL also supports a key partition. A ***key partition*** is similar to a hash partition, except the partition expression is determined automatically by the database.

Figure 38.6.2: SQL for partitions.

```
CREATE TABLE TableName
( ColumnName ColumnDefinition, ... )
[ PARTITION BY
   {   RANGE (Expression)
     | LIST (Expression)
     | HASH (Expression)
   }
   [ PARTITIONS NumberOfPartitions ]
   [ ( PartitionDefinition, ... ) ]
];

PartitionDefinition:
PARTITION PartitionName
   [ VALUES
     {   LESS THAN { (Expression) | MAXVALUE
}
       | IN ( Value, ... )
     }
   ]
   [ TABLESPACE TablespaceName ]
```


MySQL partition restrictions

MySQL has unusual restrictions that prevent partitions on many tables and columns. Ex:

- *A partitioned table may not contain foreign keys, and foreign keys may not refer to a partitioned table.*
- *All partition columns must appear in all unique columns, including the primary key, of the partitioned table.*

As a result, partitions are of limited value in MySQL.

PARTICIPATION ACTIVITY 38.6.5: Partition types.

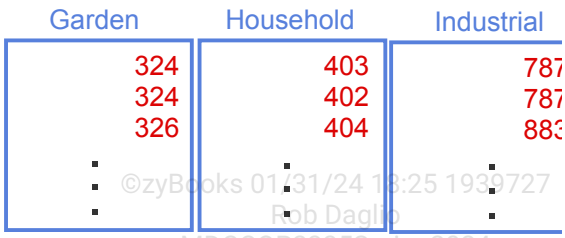

```
CREATE TABLE Order (
  OrderNumber INT,
  OrderDate DATE,
  ProductCode CHAR(3),
  OrderAmount DECIMAL(10,2)
)

PARTITION BY RANGE (YEAR(OrderDate)) (
  PARTITION p2017 VALUES LESS THAN (2018),
```

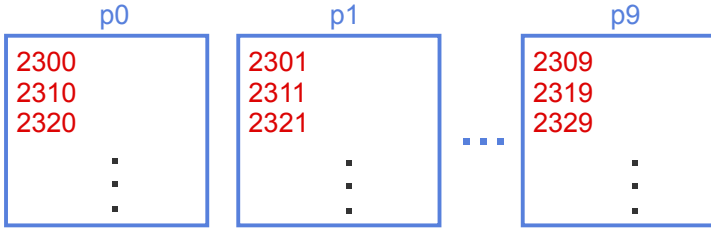
p2017	p2018	pCurrent
2014-08-14 2017-05-23	2018-01-24 2018-04-15	2019-02-22 2019-08-30

```
  PARTITION p2018 VALUES LESS THAN (2019),
  PARTITION pCURRENT VALUES LESS THAN MAXVALUE
);
```


```
PARTITION BY LIST (ProductCode) (
  PARTITION Garden VALUES IN (324, 325, 326),
  PARTITION Household VALUES IN (402, 403, 404),
  PARTITION Industrial VALUES IN (787, 883)
);
```




```
PARTITION BY HASH(OrderNumber)
  PARTITION 10;
```



Animation content:

Static figure:
A CREATE TABLE statement appears:
Begin SQL code:
CREATE TABLE Order (
 OrderNumber INT,
 OrderDate DATE,
 ProductCode CHAR(3),
 OrderAmount DECIMAL(10,2)
)
End SQL code.

Three alternative PARTITION clauses for the CREATE TABLE statement appear.
Begin SQL code.
PARTITION BY RANGE (YEAR(OrderDate)) (
 PARTITION p2017 VALUES LESS THAN (2018),
 PARTITION p2018 VALUES LESS THAN (2019),
 PARTITION pCURRENT VALUES LESS THAN MAXVALUE
);

PARTITION BY LIST (ProductCode) (
 PARTITION Garden VALUES IN (324, 325, 326),
 PARTITION Household VALUES IN (402, 403, 404),
 PARTITION Industrial VALUES IN (787, 883)
);

PARTITION BY HASH(OrderNumber)
 PARTITION 10;

End SQL code.

Next to the PARTITION BY RANGE clause, three blocks appear. The first block has caption p2017 and contains dates for 2017 and prior. The second block has caption p2018 and contains dates for 2018. The third block has caption p2019 and contains dates for 2019.

Next to the PARTITION BY LIST clause, three blocks appear. The first block has caption Garden and contains values 324 and 326. The second block has caption Household and contains values 402, 403, and 404. The third block has caption Industrial and contains values 787 and 883.

Next to the PARTITION BY HASH clause, three blocks appear. The first block has caption p0 and contains values ending in 0. The second block has caption p1 and contains values ending in 1. The third block has caption p9 and contains values ending in 9.

Step 1: Order table has columns for order number, date, product code, and amount of order. The CREATE TABLE statement appears.

Step 2: Range partition assigns partitions based on OrderDate year. The PARTITION BY RANGE clause appears.

Step 3: Rows are assigned to one of the three partitions based on year. The blocks associated with PARTITION BY RANGE appear.

Step4: List partition assigns partitions based on specific ProductCodes. The PARTITION BY LIST clause appears.

Step 5: Each partition contains rows with specific product codes. The blocks associated with PARTITION BY LIST appear.

Step 6: Hash partition expression is HASH(OrderNumber) with ten partitions The PARTITION BY HASH clause appears.

Step 7: Each row is assigned to the partition OrderNumber modulo 10, so assigned partitions are based on the order number's last digit. The blocks associated with PARTITION BY HASH appear.

Animation captions:

1. Order table has columns for order number, date, product code, and amount of order.
2. Range partition assigns partitions based on OrderDate year.
3. Rows are assigned to one of the three partitions based on year.
4. List partition assigns partitions based on specific ProductCodes.
5. Each partition contains rows with specific product codes.
6. Hash partition expression is HASH(OrderNumber) with ten partitions.

7. Each row is assigned to the partition OrderNumber modulo 10, so assigned partitions are based on the order number's last digit.

PARTICIPATION ACTIVITY 38.6.6: Partitions.

Product codes are positive integers. Products are grouped by product code as follows:

Group	ProductCode
1	0 to 3999
2	4000 to 8499
3	8500 to 9199
4	9200 and above

The Order table has 200 million rows. Queries usually specify product code and access orders for several products within one group. To improve query performance, the Order table is partitioned as follows:

```
CREATE TABLE Order (
   OrderNumber INT,
   OrderDate DATE,
   ProductCode SMALLINT UNSIGNED,
   Amount DECIMAL(10,2)
)
PARTITION BY ___A___ ( ___B___ )   (
   PARTITION p1 VALUES LESS THAN (4000),
   PARTITION p2 VALUES LESS THAN (___C___),
   PARTITION p3 VALUES LESS THAN (___D___),
   PARTITION p4 VALUES LESS THAN ___E___
);
```

1) What is keyword A?

Check Show answer

2) What is column B?

Check Show answer

3) What is value C?

Check Show answer

4) What is value D?

Check Show answer

5) What is keyword E?

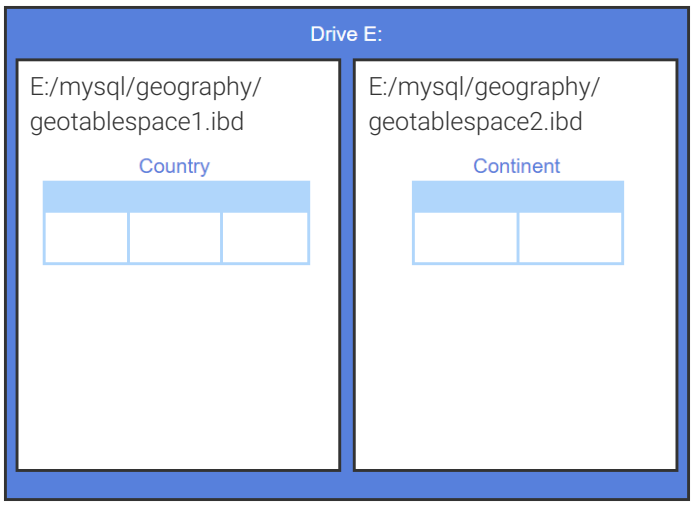
Check Show answer

CHALLENGE ACTIVITY 38.6.1: Tablespaces and partitions.

539740.3879454.qx3zqy7

Start

Continent and Country are large tables frequently updated, and therefore assigned to different



Complete the values of A, B and C:

(A) /* Type your code here */

```
CREATE TABLESPACE GeoTable
ADD DATAFILE
    'E:/mysql/geography/geo

CREATE __(A)__ GeoTablespac
__(B)__
    'E:/mysql/geography/geo

CREATE TABLE Country (
    Code SMALLINT,
    Name VARCHAR(15),
    Capital VARCHAR(15)
)
TABLESPACE GeoTablespace1;

CREATE TABLE Continent (
    Code CHAR(2),
    Name VARCHAR(15),
```

(B)

(C)

```
)
TABLESPACE __(C)__;
```

1 2 3 4

Check Next

Exploring further:

- MySQL tablespace documentation
- MySQL partition documentation

38.7 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.

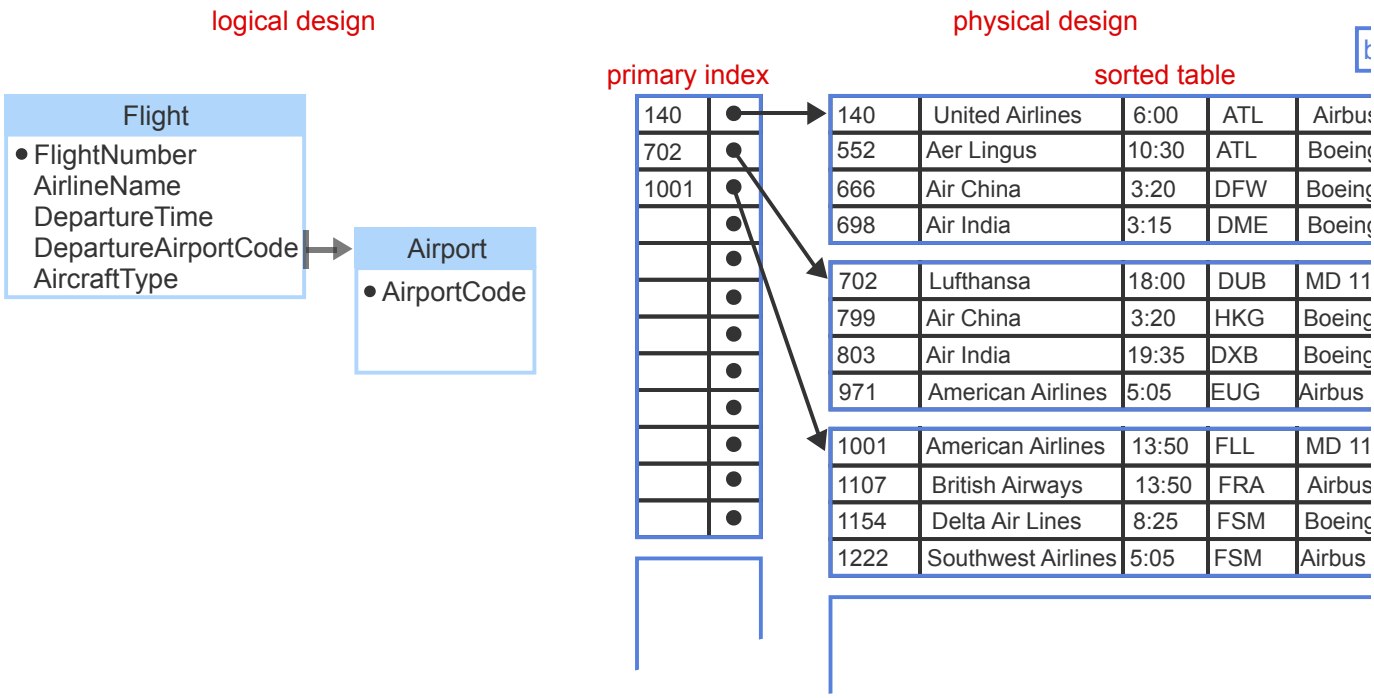
Different databases and storage engines support different table structures and index types. Ex:

- *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 38.7.1: 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 38.7.2: 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 38.7.1: 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 38.7.2: SHOW INDEX result table (selected columns).

Column Name	Column Meaning

Table	Name of indexed table
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 38.7.3: 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.
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 38.7.4: 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 38.7.3: 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
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 38.7.5: EXPLAIN statement.

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

(57 rows total)


```
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	filtere
SIMPLE	Address	ALL	idx_fk_city_id, PostalCodeIndex	NULL	NULL	603	33.33
SIMPLE	City	eq_ref	PRIMARY	PRIMARY	sakila.address.city_id	1	100.0

(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

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 38.7.6: 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 ACTIVITY 38.7.7: 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
    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 38.7.8: 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

38.8 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 and is slow.*

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 is fast.*

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. Since the query has > in the WHERE clause rather than =, the query executes a table scan and is slow.*

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 and is slow.*

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 uses an index scan, which is faster than a table scan (step 5).

For submit-mode testing, all seven statements must appear in Main.sql in the correct order.

NOTES:

- `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.
- If you try this lab in MySQL Workbench, drop the index `idx_title` from `film` prior to executing statement 1.
- In submit-mode tests that generate multiple result tables, the results are merged. Although the tests run correctly, the results appear in one table.

LAB ACTIVITY | 38.8.1: LAB - Create index and explain (Sakila) | 0 / 10

Main.sql — Load default template...

```
1 Loading latest submission...
```


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?

```
History of your effort will appear here once you begin working
on this zyLab.
```


38.9 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:

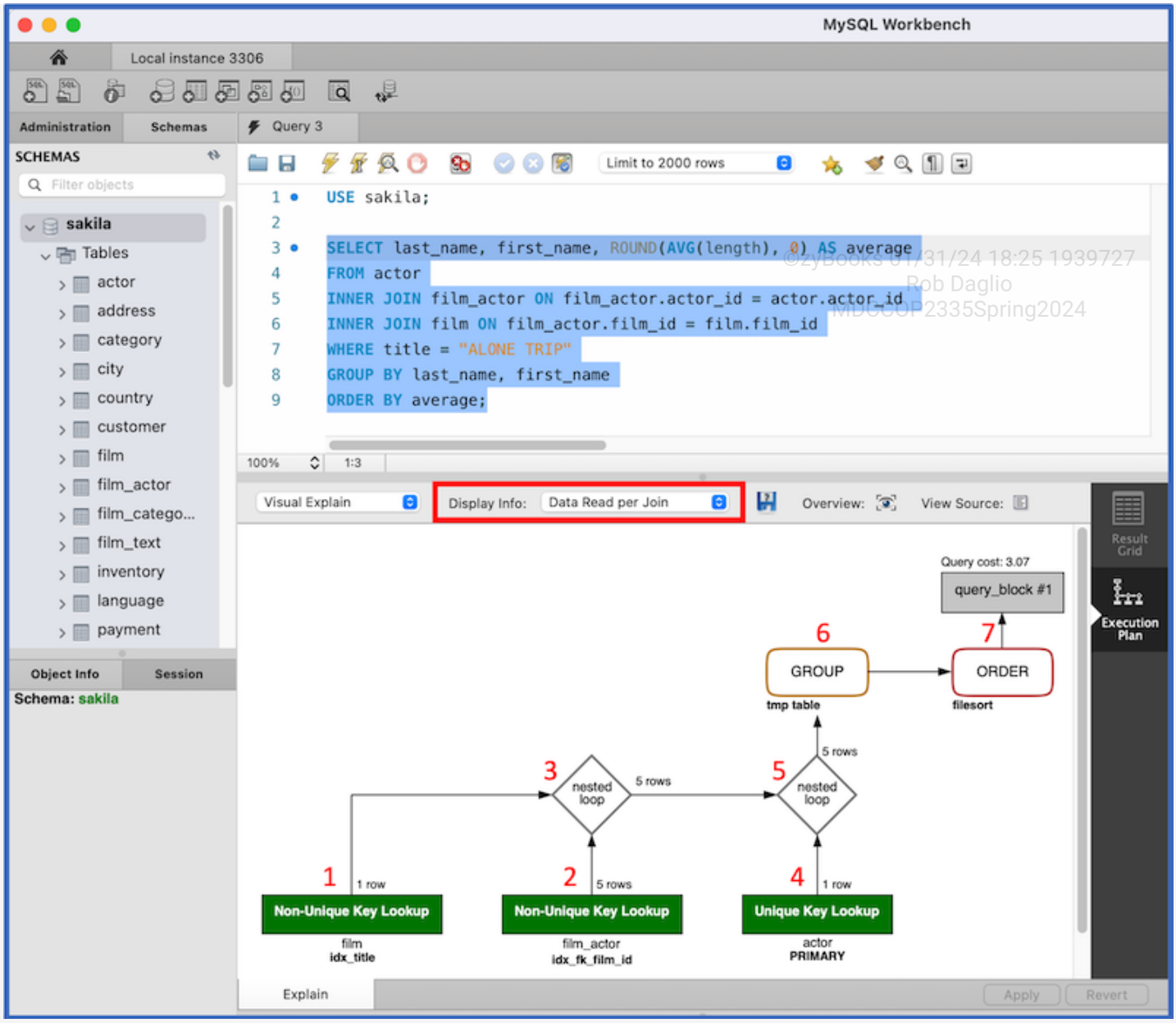
1. Enter the following statements:

```
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.

LAB ACTIVITY | 38.9.1: LAB - Query execution plans (Sakila) 0 / 10

Main.sql

```
1 Loading latest submission...
```

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?

Retrieving signature

©zyBooks 01/31/24 18:25 1939727
Rob Daglio
MDCCOP2335Spring2024