13.1 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 13.1.1: Media characteristics (circa 2018).

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

memory				
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 13.1.1: 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

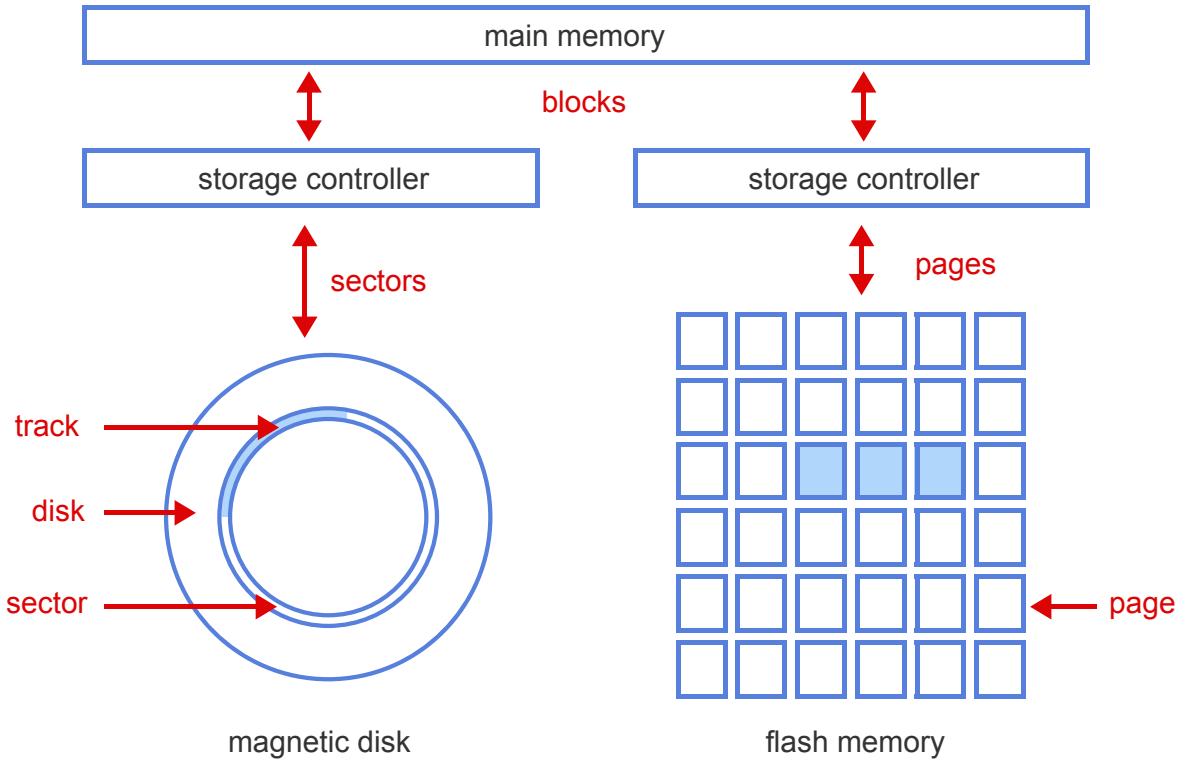
Magnetic disk groups data in ***sectors***, traditionally 512 bytes per sector but 4 kilobytes with newer disk formats. Flash memory groups data in ***pages***, usually between 2 kilobytes and 16 kilobytes per page.

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 is usually configurable when the database system is installed.

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 13.1.2: Sectors, pages, and blocks.



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

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 13.1.3: 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***.

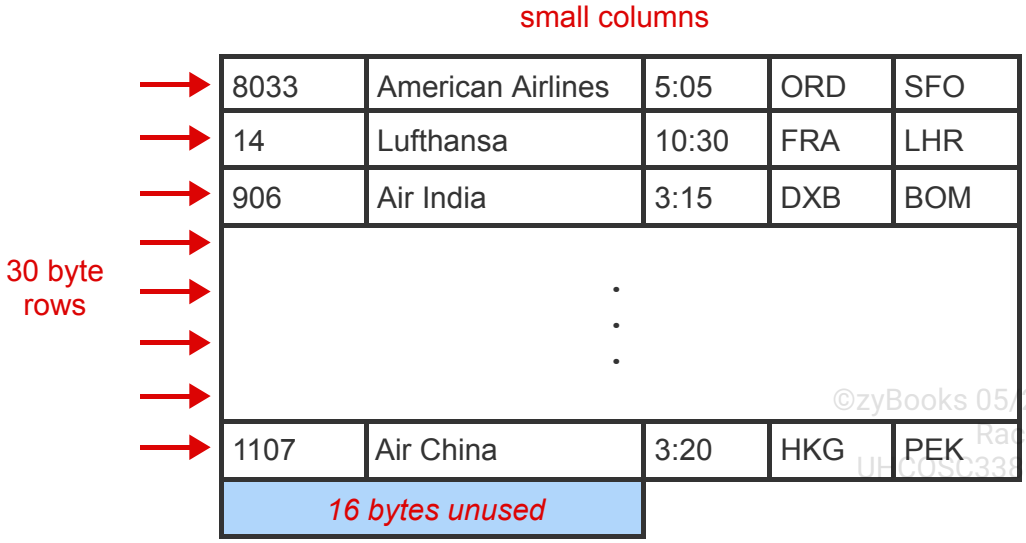
Row-oriented storage performs best when row size is small relative to block size, for two reasons:

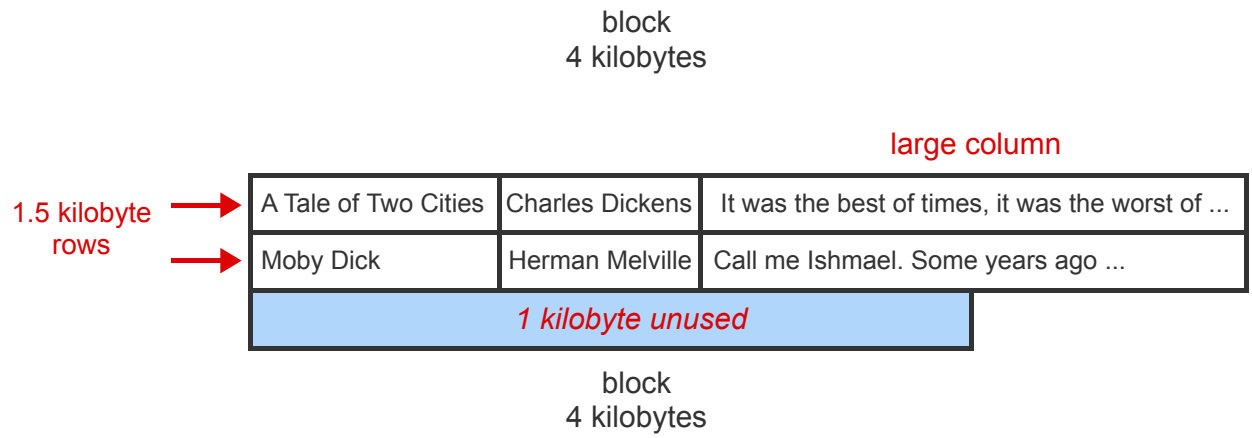
- *Improved query performance.* When row size is small relative to block size, each block contains many rows. Queries that read and write multiple rows transfer fewer blocks, resulting in better performance.
- *Less wasted storage.* Row-oriented storage wastes a few bytes per block, since rows do not 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 13.1.4: 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.

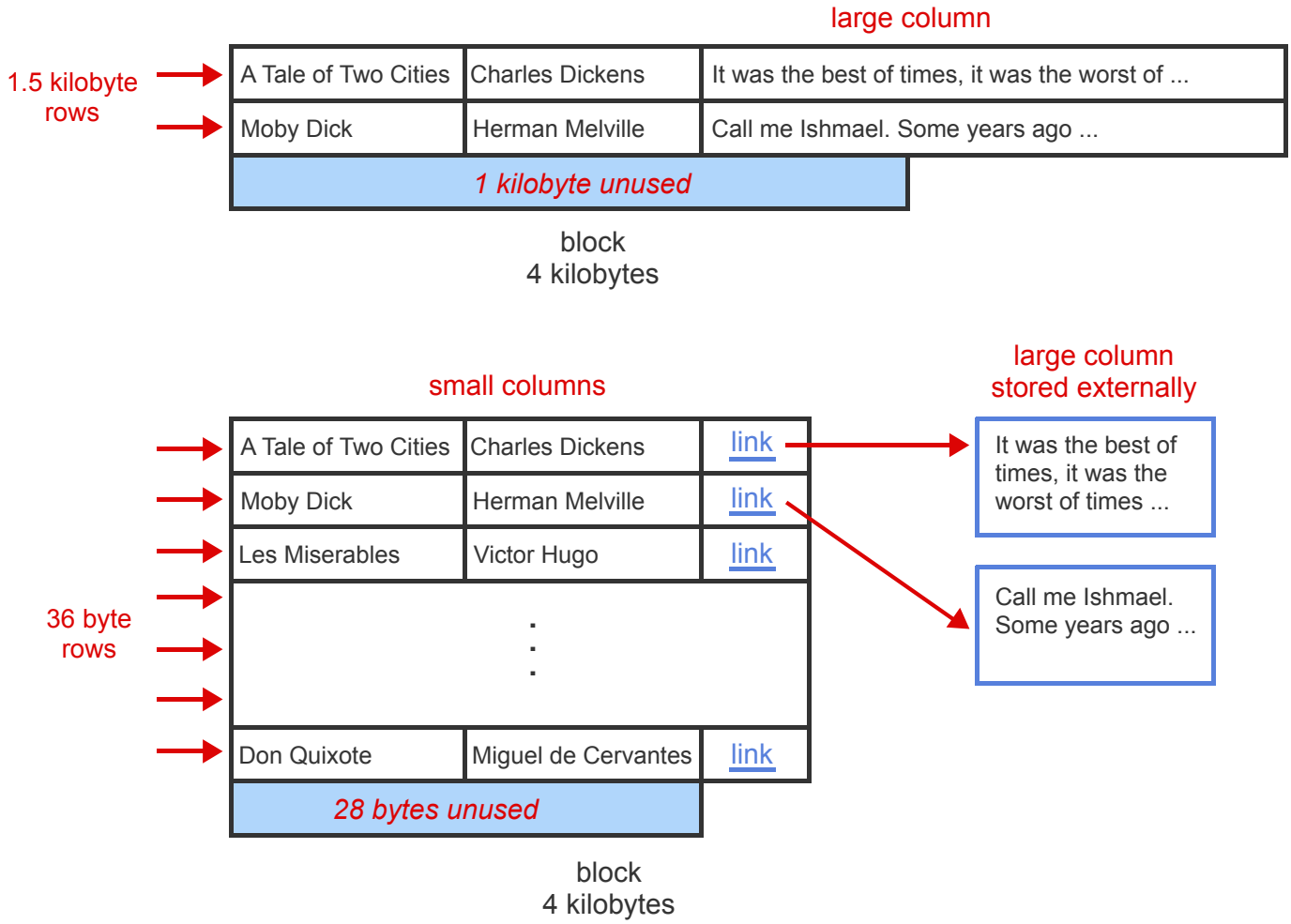
Step 3: The row size is large relative to block size, so fewer rows are transferred per block. Another 4 kilobyte block appears with two data rows and one empty row. Each data row contains book title, book author, and book text. The book text column has caption large column.

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 13.1.5: Row-oriented storage with large columns.



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 13.1.6: 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.

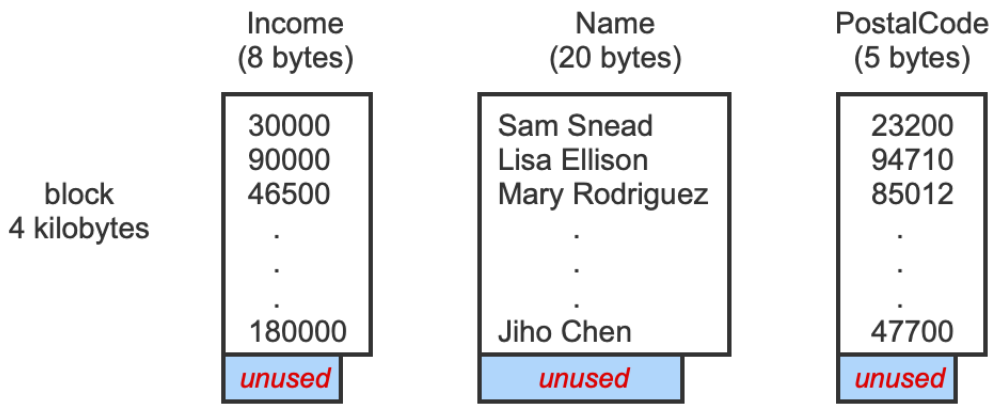
Column-oriented storage benefits analytic applications in several ways:

- *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 13.1.7: 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 | 13.1.8: 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?

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

13.2 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

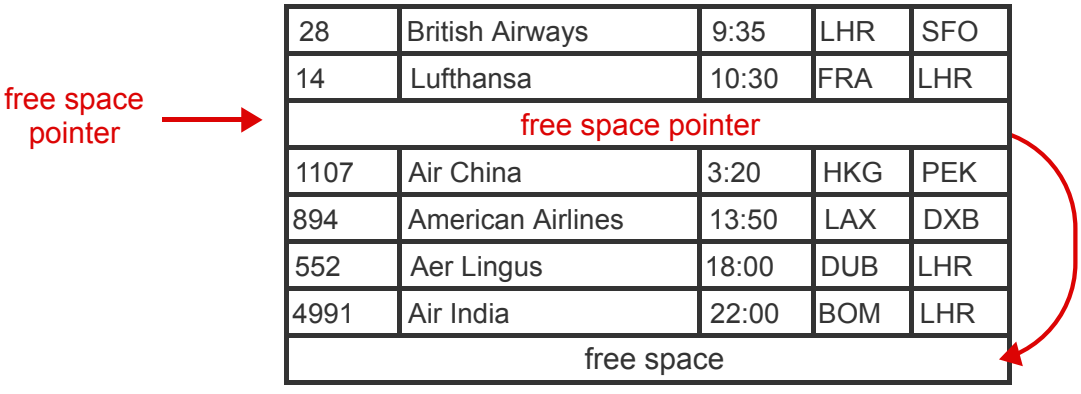
Each table in a database can have a different structure. Databases assign a default structure to all tables. Database administrators can override the default structure to optimize performance for specific queries.

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 13.2.1: 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.

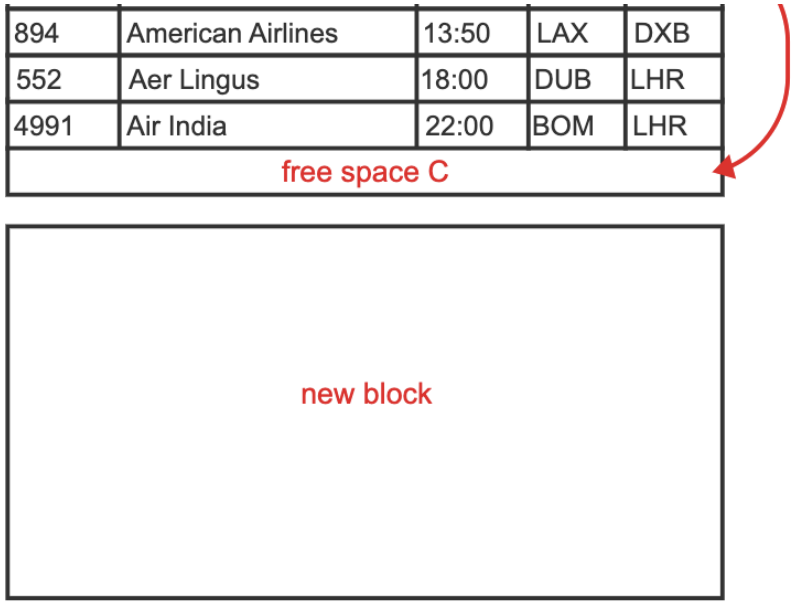
Step 4: As more rows are deleted, free space is linked together in a list. Another row is deleted that comes before in the table. The free space pointer moves to this deleted space and the deleted space is linked to the previously deleted row.

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 13.2.2: 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 A** **Free space B** **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 13.2.3: 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 | 13.2.4: 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.

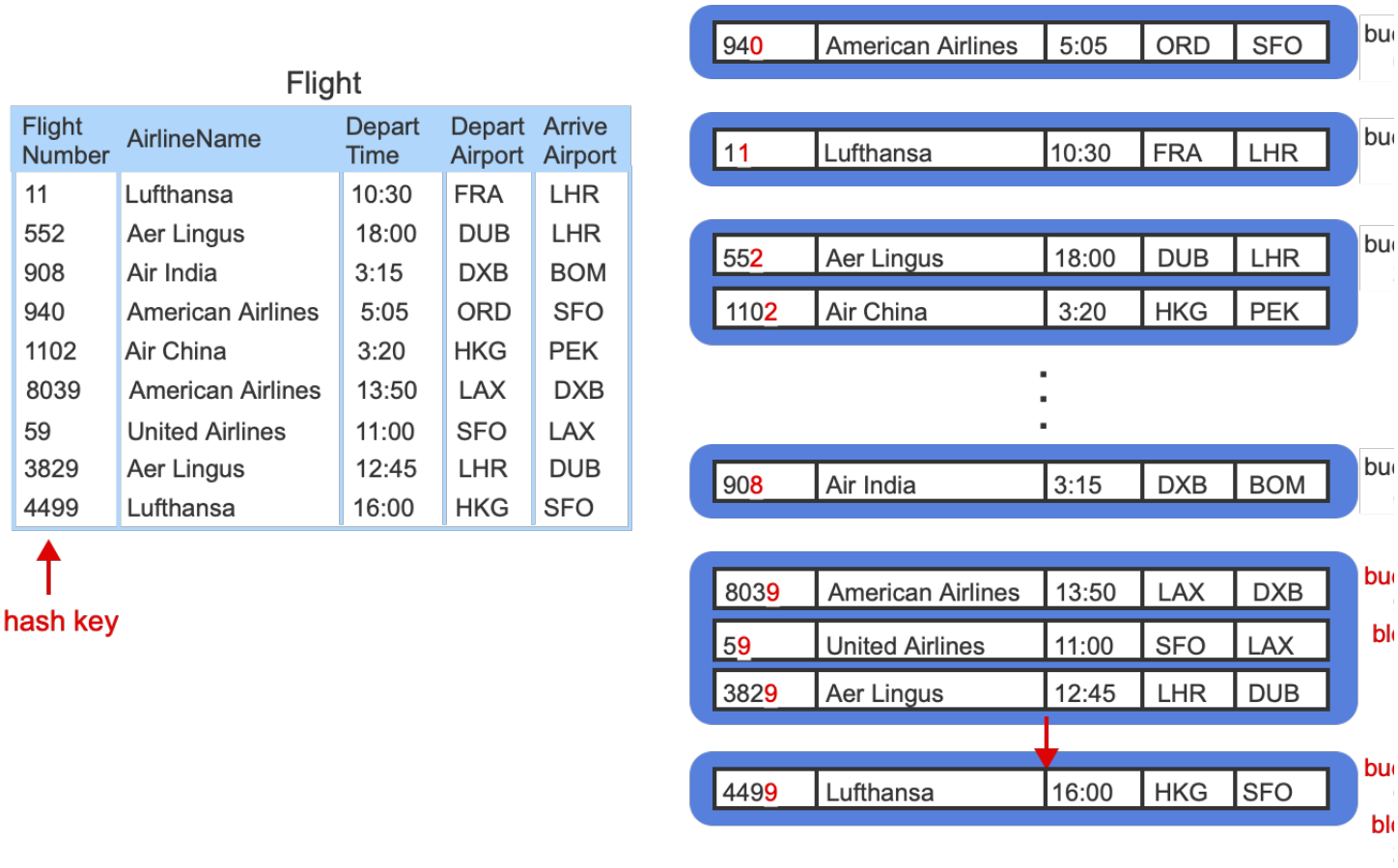
Hash functions are designed to scramble row locations and evenly distribute rows across blocks. The ***modulo function*** is a simple hash function with four steps:

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 | 13.2.5: Assigning rows to buckets with a modulo function.



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.

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 13.2.6: 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

- 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 13.2.1: Table structures.

544874.3500394.qx3zqy7

Start

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

252	Darcy	KY
1967	Rhys	AK
310	Enrique	AZ
999	Shayan	WA
907	Akunna	TN
1001	Niko	OR
free space A		

Where does the free space pointer point to?

Select

Row 999 is deleted and becomes free space B. What does the free space linked list look lil

Select

1 2 3 4 5

Check Next

Table clusters

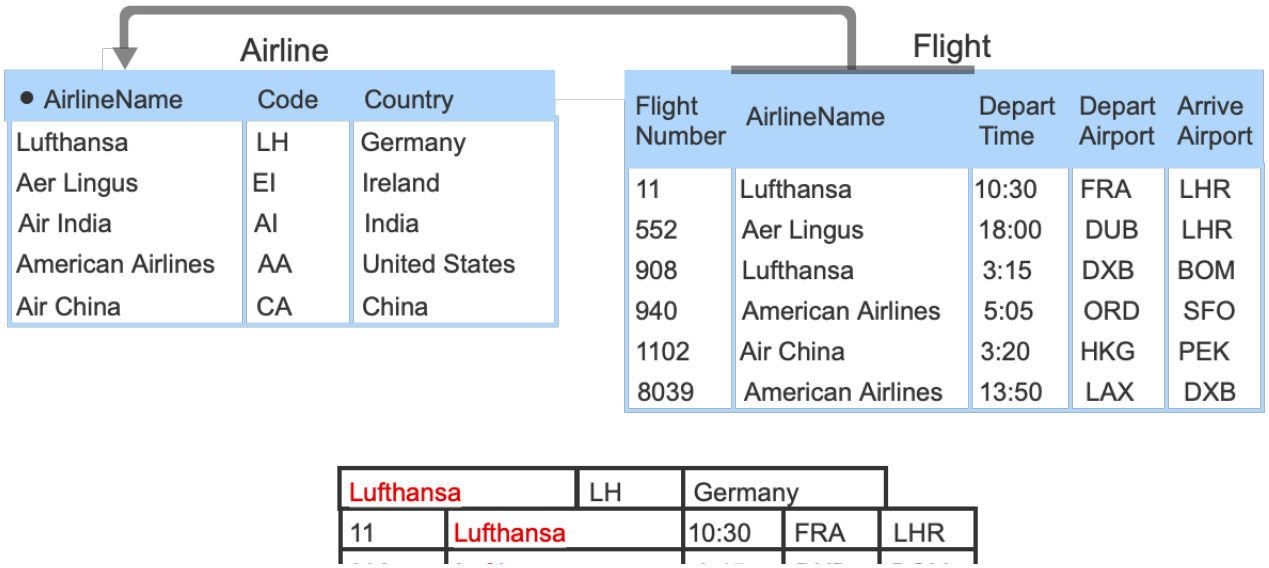
Table clusters, also called ***multi-tables***, interleave rows of two or more tables in the same storage area. Table clusters have a ***cluster key***, a column that is available in all interleaved tables. The cluster key determines the order in which rows are interleaved. Rows with the same cluster 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 13.2.7: Table cluster.



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 | 13.2.8: 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

13.3 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 13.3.1: 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 13.3.1: Assigning three tables to the same tablespace.

Animation captions:

PARTICIPATION ACTIVITY | 13.3.2: 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

5) 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.

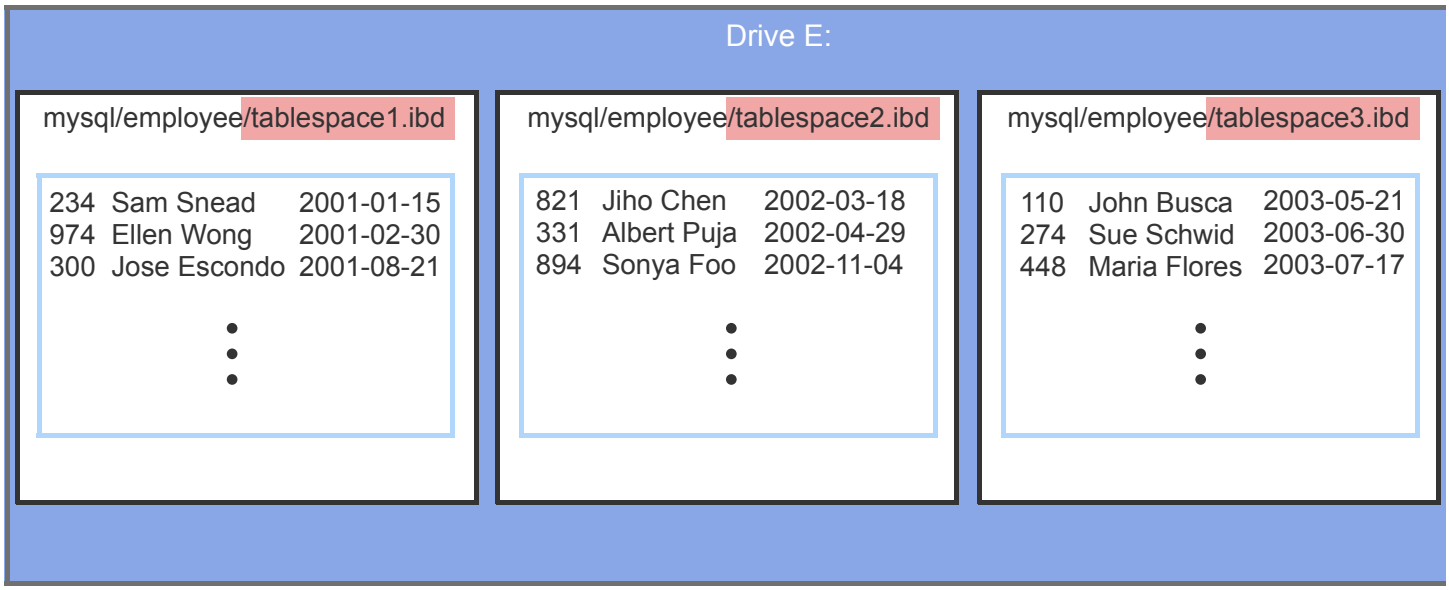
Partitions improve query performance by reducing the amount of data accessed by INSERT, UPDATE, DELETE, and SELECT statements. Ex: The Sales table contains sales transactions for the past ten years, with one partition for each year. Most queries access current year sales only. The current-year partition 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 | 13.3.3: 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 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 13.3.4: 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 13.3.2: 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 13.3.5: Partition types.

Animation captions:

PARTICIPATION ACTIVITY 13.3.6: 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?

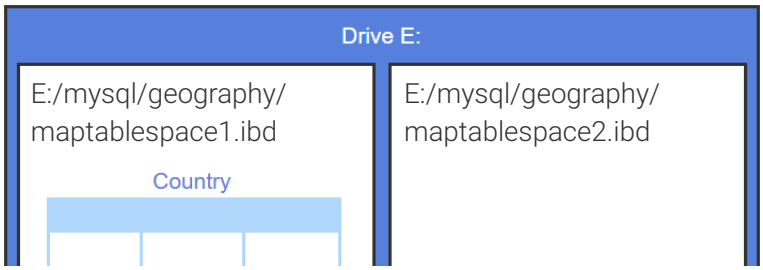
Check Show answer

CHALLENGE ACTIVITY 13.3.1: Tablespaces and partitions.

544874.3500394.qx3zqy7

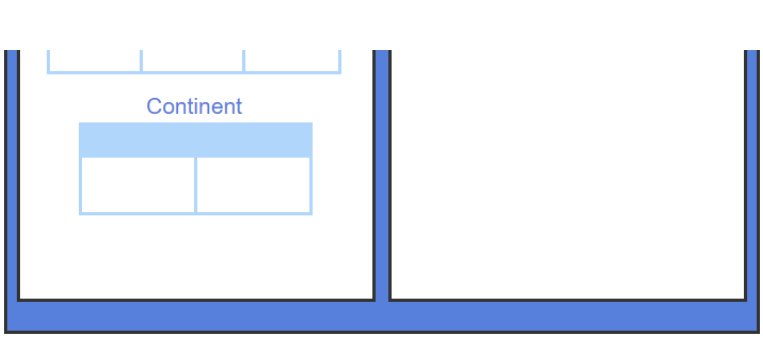
Start

Continent and Country are small, read-only tables, and therefore assigned to the same tab




```
CREATE __(A)__ MapTables
ADD DATAFILE
    'E:/mysql/geography/

CREATE TABLESPACE MapTa
  (B)
```



Complete the values of `A`, `B` and `C`:

(A) /* Type your code here */

(B)

(C)

```
__`)__
    'E:/mysql/geography/

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

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

1 2 3 4

Check Next

Exploring further:

- MySQL tablespace documentation
- MySQL partition documentation