5.4 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.

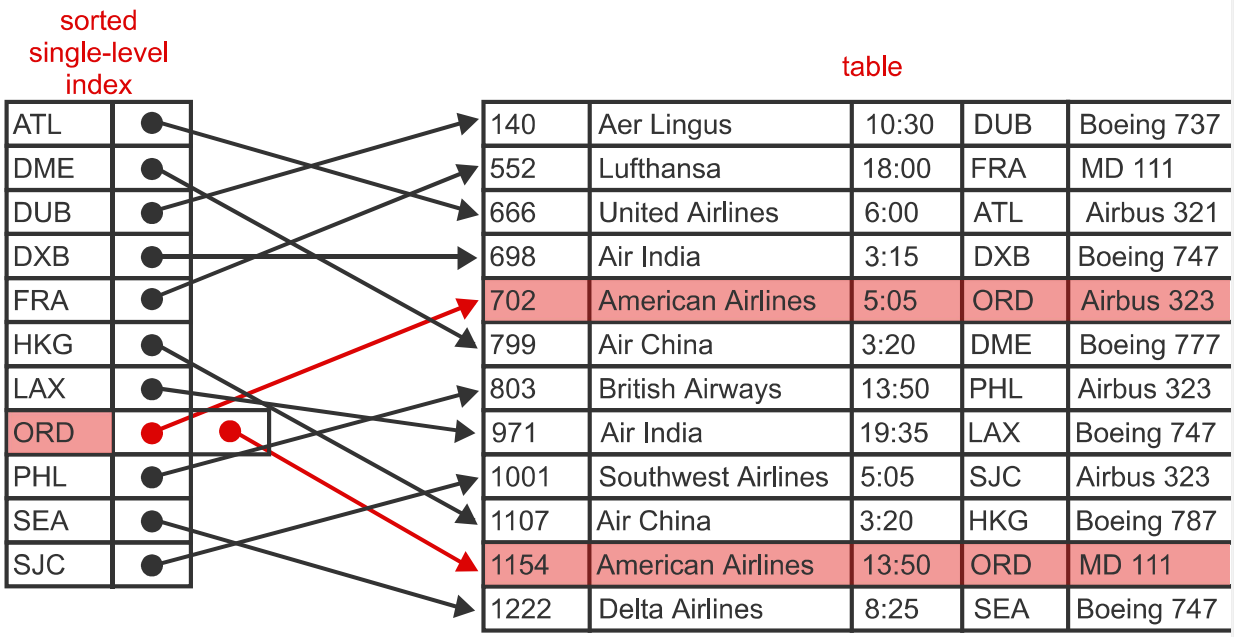
Single-level indexes are normally sorted on the column value. A sorted index is not the same as an index on a sorted table. Ex: An index on a heap table is a sorted index on an unsorted table.

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 5.4.1: Single-level index.

1 2 3 4 5 2x speed



Alternatively, an index on a non-unique column may have multiple pointers for some values

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.

Feedback?

PARTICIPATION ACTIVITY 5.4.2: 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
- (selected) The sixth entry
- The eleventh entry

Correct

The sixth entry contains the value HKG, with a pointer to the row for Air China flight 1107.

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.
- (selected) Between the LAX and ORD index entries.

Correct

The index is on the DepartureAirport column. Since the index is sorted, an entry for MUC goes between LAX and ORD. The database must create space for the new entry.

3) An indexed column requires 12 bytes. Pointers to table blocks

Correct

Each row requires one index entry. Each index entry is 12 + 8 bytes = 20 bytes. A 4 kilobyte block can contain about

require 8 bytes. Index blocks are 4 kilobytes. How many rows can be referenced in one index block?

- (•) 200
- ( ) 400
- ( ) Unlimited

4,000 / 20 = 200 entries.

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.

Correct

Strategy A does not repeat column values, so indexes are more compact.

Feedback?

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:

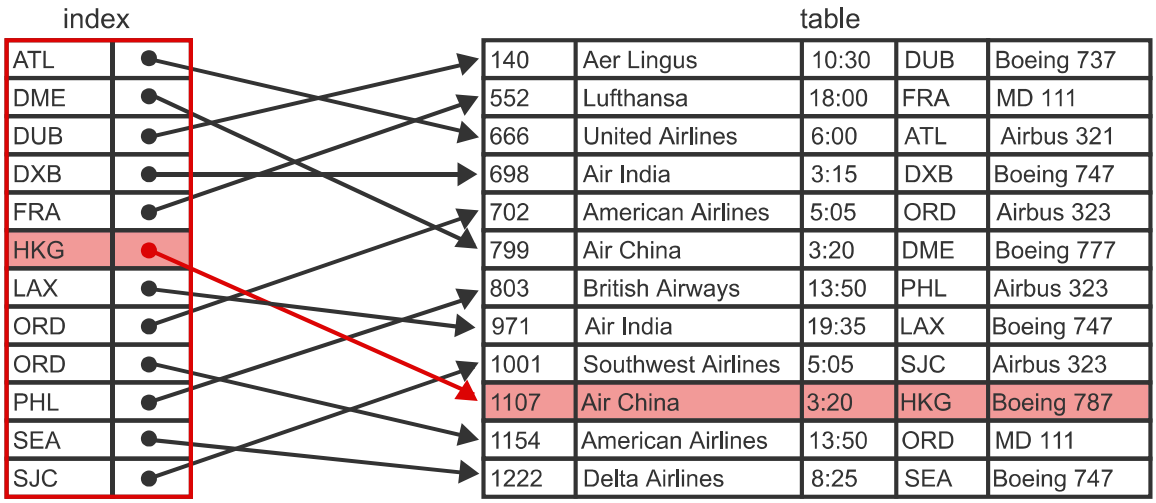
1. Looks for an indexed column in the WHERE clause.
2. Scans the index.
3. Finds values that match the WHERE clause.
4. Reads the corresponding table blocks.

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 5.4.3: Query processing with single-level index.

1 2 3 4 5 2x speed




```
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

The WHERE clause contains an indexed column. Since the hit ratio is low, the database performs a scan.

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.

Feedback?

PARTICIPATION ACTIVITY 5.4.4: 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

Correct

The database reads at most 200 index blocks, plus one table block containing the row for flight 3988.

2) What is the minimum number of blocks necessary to process the SELECT?

- 1
- (•) 2
- 201

Correct

The database performs an index scan. If the entry for flight 3988 is in the first block of the index, the database reads one index block and one table block.

3) Assume the table has no index. What is the maximum number of blocks necessary to process the SELECT?

- 1
-

Correct

The database performs a table scan and reads at most all 2,000 table blocks.

201
2,000

Feedback?

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:

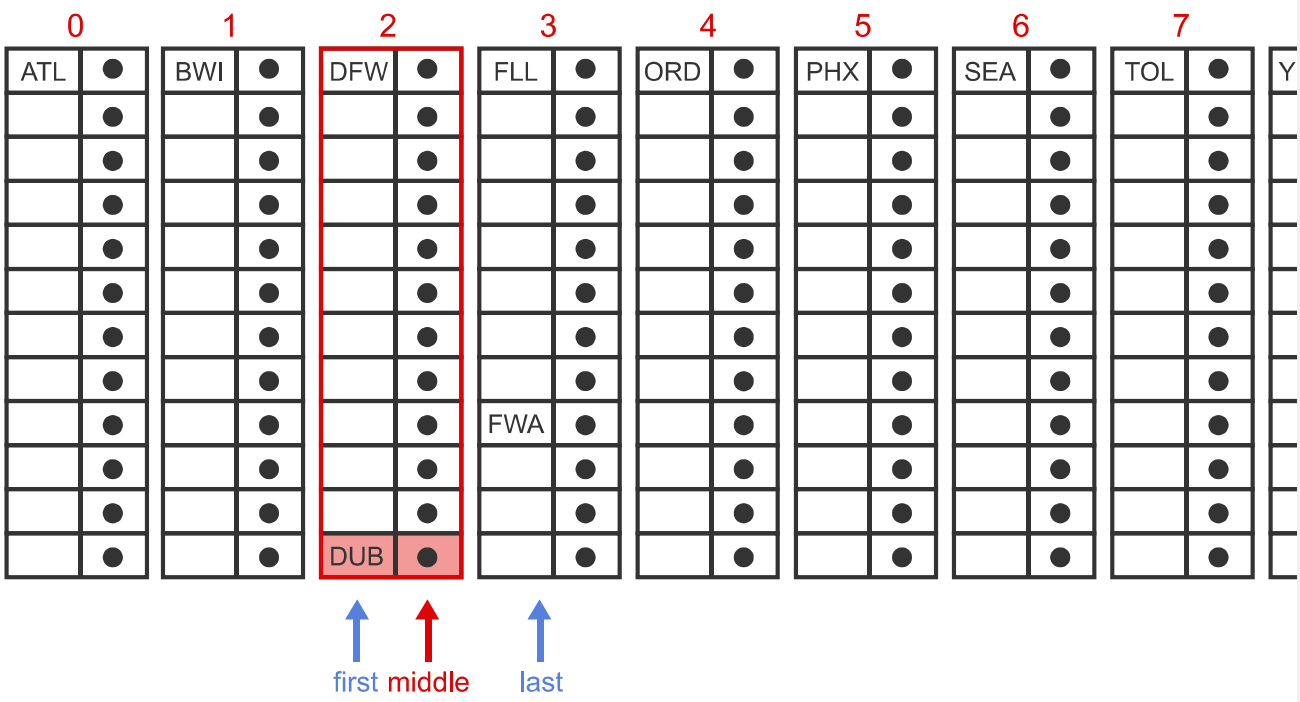
1. The database first compares the search value to an entry in the middle of the index.
2. If the search value is less than the entry value, the search value is in the first half of the index. If not, the search value is in the second half.
3. The database now compares the search value to the entry in 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 entries, a binary search examines $\log_2 N$ values, rounded up to the nearest integer. In the example above, the index has 10 million entries. The binary search reads at most $\log_2 10{,}000{,}000 = 24$ blocks and requires about 0.001 seconds (24 blocks × 4 kilobytes/block / 0.1 gigabytes/sec).

PARTICIPATION

ACTIVITY 5.4.5: Binary search on a sorted index.

1 2 3 4 5 2x speed




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

The binary search reads block 2 and finds 'DUB'. The database follows DUB's pointer to the table containing the row.

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.

Feedback?

PARTICIPATION ACTIVITY 5.4.6: 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

Correct

100,000,000 rows × 400 bytes/row / 0.1 gigabytes/sec = 400 seconds.

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

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

Correct

Each index block contains 8,000 bytes / 20 bytes/entry = 400 entries. The index has 100,000,000 entries / 400 entries/block = 250,000 blocks. A binary search requires log 250,000 (base 2).

Feedback?

Primary, clustering, and secondary indexes

Indexes on a sorted table may be primary, clustering, or secondary:

- A ***primary index*** is an index on a unique sort column.
- A ***clustering index*** is an index on a non-unique sort column.
- A ***secondary 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 or clustering index. 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 and clustering 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:

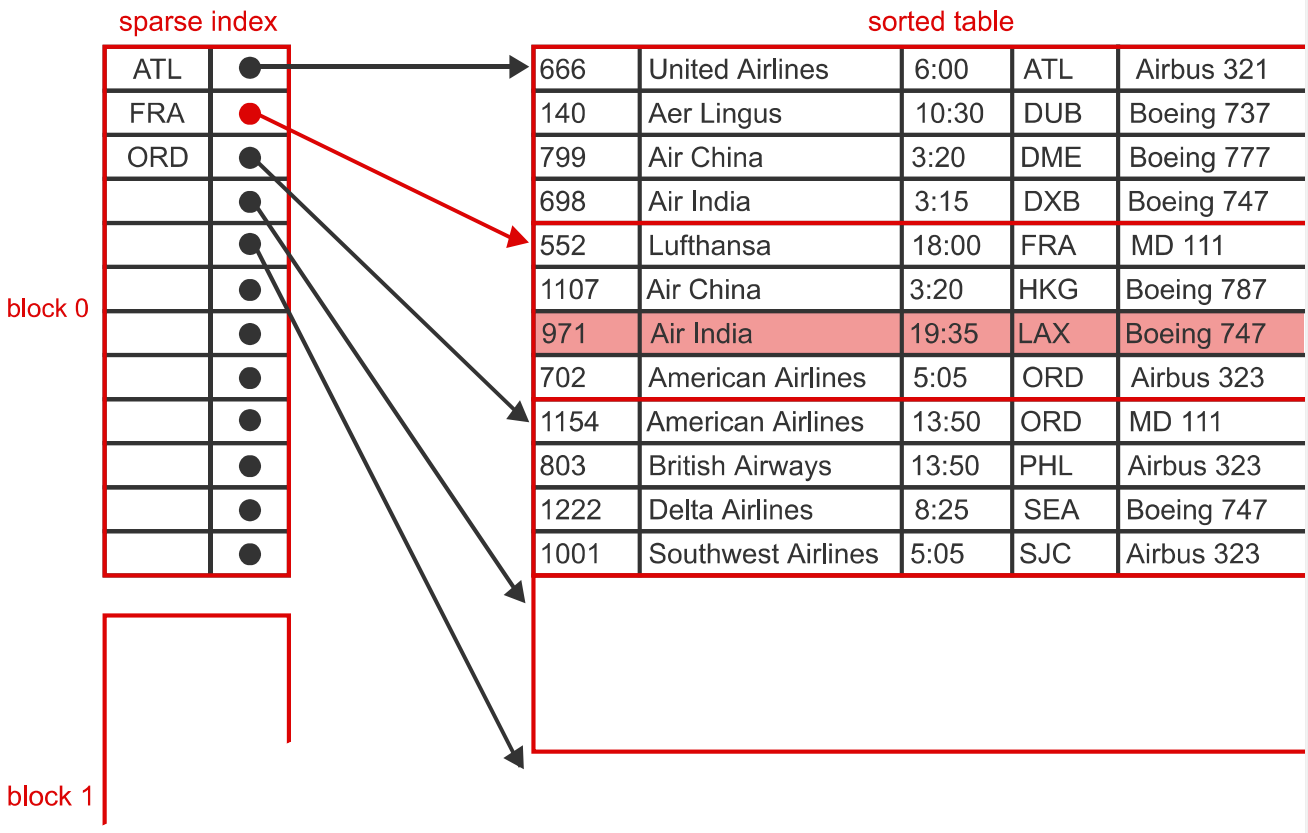
- A table has 10 million rows.
- Each row is 100 bytes.
- Table and index blocks are 4 kilobytes.
- Each index entry is 10 bytes.

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 and clustering indexes are usually sparse and sparse indexes are fast. As a result, database designers usually create a primary or clustering index on large tables.

PARTICIPATION ACTIVITY 5.4.7: Dense and sparse indexes.

1 2 3 4 2x speed



'LAX' does not appear in the index, but the block containing 'LAX' can be determined from sort c

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.

Feedback?

PARTICIPATION ACTIVITY 5.4.8: Primary, clustering, and secondary indexes.

Match the term with the term's description.

Dense index	Index on any column of the table, including sort column, with one entry for each row. A dense index has one entry for each table row. Dense indexes can be created on any column, but dense indexes are not usually created on sort columns.	Correct
Sparse index	Index on either a unique or non-unique table sort column. A sparse index is always on the sort column and contains one entry for each table block.	Correct
Primary index	An index on the primary key of a table. A primary index is an index on a unique column. Usually, a primary index is on the table's primary key.	Correct

Clustering index	An index on a non-unique sort column. A clustering index is an index on a sort column that is not unique. Since a table can only be sorted on one column, a table has at most one clustering or primary index.	Correct
Secondary index	An index not on the table sort column. A secondary index is an index on a column other than the sort column. Secondary indexes must be dense. A table can have multiple secondary indexes.	Correct

Reset

Feedback?

Inserts, updates, and deletes

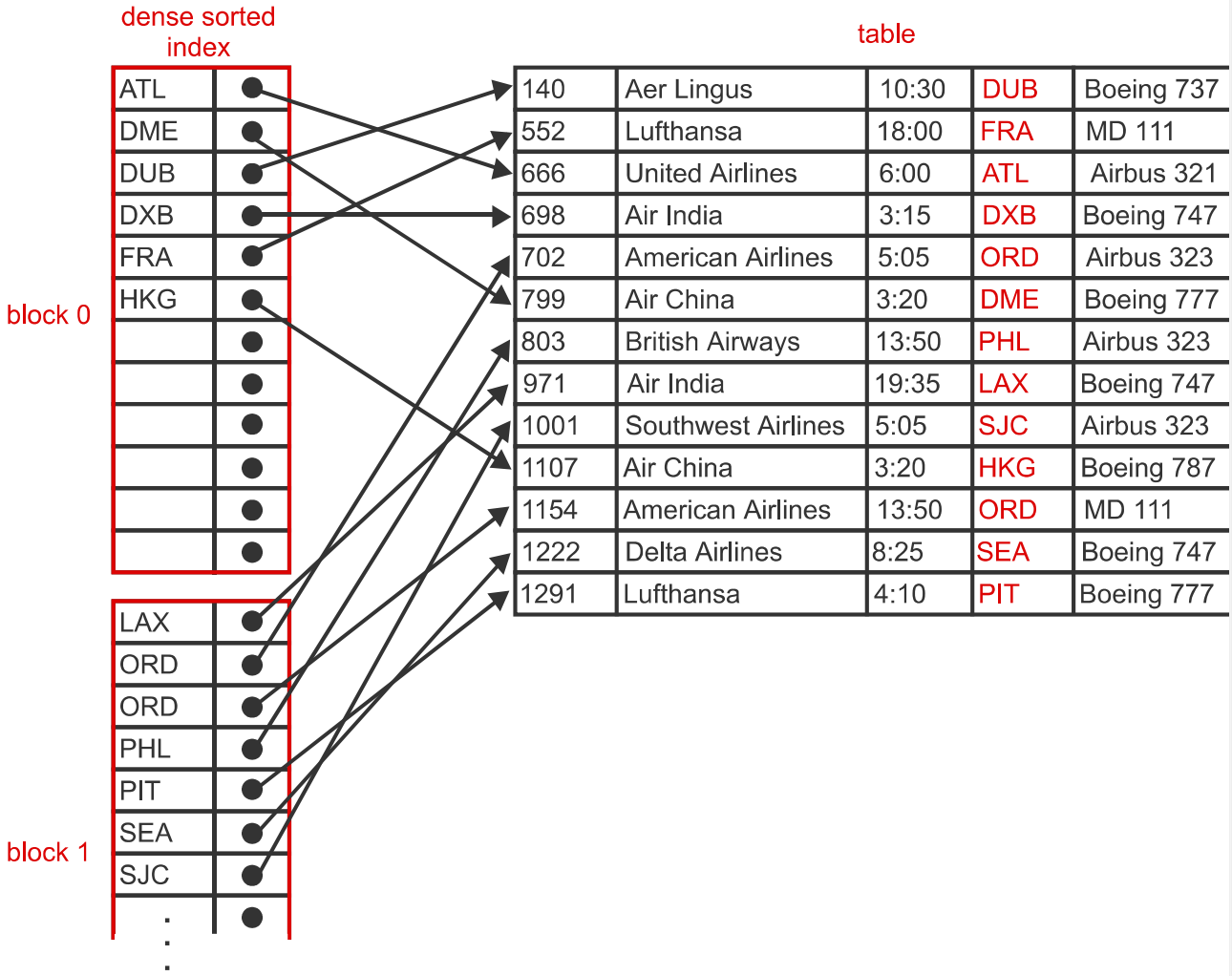
Inserts, updates, and deletes to tables have an impact on single-level indexes. Consider the behavior of dense indexes:

- **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 5.4.9: Insert with dense sorted index.

1 2 3 2x speed



The block split creates space for the new index entry.

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.

Feedback?

PARTICIPATION ACTIVITY 5.4.10: Inserts, updates, and deletes.

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

- ○ True
- ◉ False

Correct

Inserting a row may generate a new index entry, which slows down the insert. However, if the index is on the sort column, the index helps locate the table block for the insert, which speeds up the query.

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

- ◉ True
- ○ False

Correct

In a heap table, inserts always go to the end of the table, so indexes do not help. Since inserts always require new entries to a dense index, and secondary indexes are always dense, secondary indexes on heap tables always slow inserts.

3) Most tables have a primary or clustering index.

- ◉ True
- ○ False

Correct

Primary and clustering indexes are sparse and therefore compact and fast. Updates to primary and clustering indexes only occur when table blocks are added or deleted, which is infrequent. Consequently, primary and clustering indexes usually improve performance and are created on most tables.

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

- ◉ True
- ○ False

Correct

An update on a column with a dense index deletes the old index entry and inserts a new index entry. If the index block for the new entry has no free space, the block splits.

Feedback?

CHALLENGE ACTIVITY 5.4.1: Single-level indexes.

379958.2369558.qx3zqy7

Jump to level 1

1 2 3 4 5

Consider the following scenario:

- A table has 900,000,000 rows.
- Each index block is 2 kilobytes.

- Each index entry is 100 bytes.
- Assume 1 kilobyte is approximately 1,000 bytes.

6

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

$\log_2$ 45000000 blocks

1 2 3 4 5 **6**

Check Next **Done**. Click any level to practice more. Completion is preserved.

Expected: $\log_2$ 45000000 blocks

Each index block contains 2 kilobytes × (1,000 bytes/kilobyte) / 100 bytes/entry = 20 entries.
The index has (900,000,000 entries) / (20 entries/block) = 45,000,000 blocks.
A binary search requires $\log_2$ 45,000,000 blocks (rounded up to the nearest integer).

Feedback?

How was this section? **Provide feedback**