Indexes

Markings: PK - primary key; FK - foreign key

Creating B+ tree index

```
                  unique index   (S; clustered index)                                an index can be named
                                                             (PL)                    e.g. in this way:
PMOSL  CREATE [UNIQUE] [CLUSTERED] INDEX [IF NOT EXISTS] index1  <-                  table1_col1_[col2_]idx
                                       ASC is default
                                                          (P)
       ON table1 ({col1[(n)] | (expr1)} [DESC] [NULLS {LAST|FIRST}] [,...])
          (M; prefix index)    index on expression (S: no)             multi-column index
                               (O: aka. function-based index)          aka. composite index

(PS)   [INCLUDE (coli [,...])]          <- covering index
(PO)   [TABLESPACE tablespace1]         <- see doc; (O: tablespace1 can be DEFAULT)
(PSL)  [WHERE filter-predicate]         <- partial index (S: aka. filtered index)
```

Creating full text, spatial, and other indexes (see a documentation for details)

```
P     CREATE INDEX index1 ON table1 USING {GIST|SPGIST|GIN|BRIN} (...);
 O    CREATE INDEX index1 ON table1 (col1) INDEXTYPE IS {MDSYS.SPATIAL_INDEX|CTXSYS.CONTEXT};
 O    CREATE BITMAP INDEX index1 ON table1 (col1 [,...]);
M     CREATE {FULLTEXT|SPATIAL}                    INDEX ...;   - spatial index uses R-tree
  S   CREATE {FULLTEXT|SPATIAL|COLUMNSTORE} INDEX ...;   - COLUMNSTORE - in RAM, good for warehouses
```

Creating index together with table

- every PRIMARY KEY and UNIQUE constraint has an automatically created index

```
        CREATE TABLE table1 (
P  O      after a definition of a PRIMARY KEY or UNIQUE table/column constraint, you can specify options for the
          automatically created index: (P: [INCLUDE (coli [,...])]) (PO: [USING INDEX TABLESPACE tablespace1])

M  S      outside of a column definition:
            S:  {PRIMARY KEY | UNIQUE |                  INDEX index1} [[NON]CLUSTERED] ( col1              [DESC] [,...])
                                          FULLTEXT
            M:  {PRIMARY KEY | UNIQUE | [ SPATIAL  ] INDEX index1}                ({col1[(n)] | (expr1)} [DESC] [,...])
                                                      |||
                                                      KEY
   S      in a column definition:
            col1 T1 {PRIMARY KEY | UNIQUE | INDEX index1} [[NON]CLUSTERED]
        )
 O      [ORGANIZATION INDEX [overflow-options]]   <- (O; clustered index; aka.
                                                      index-organized table)
```

Hiding index

- invisible index is an index that cannot be used by the optimizer, but is still maintained during DML statements
- it is useful for testing the effect of removing an index on query performance without actually removing the index

```
           (M)
MO    ALTER TABLE table1 ALTER INDEX index1 [IN]VISIBLE
P     (possible by using an extension)
```

Deleting index

```
                     (PSL)              (MS)
PMOSL  DROP INDEX [IF EXISTS] index1 ON table1     - (S: see also "DROP FULLTEXT INDEX ON table1")
```

Usage

Indexes are used to:

- enforce a unique constraint,
- improve performance of: JOIN, WHERE (comparison for =, <, >), ORDER BY, and GROUP BY clauses.

But each index decreases insert/update/delete performance, so you should have them where they're actually needed.

Every table should have a PK; every PK and UNIQUE constraint has an index.
FK generally should have an index (MySQL creates it automatically for every FK).

B+ tree index options

Unique index (PMOSL)

- values in the indexed column(s) are unique
- created automatically for each PRIMARY KEY and UNIQUE constraint
- sometimes it makes sense to create a unique index explicitly, e.g. on the UPPER(col1) expression or create a unique partial index

Multi-column index (PMOSL)

- indexes more than one column of a table
- the order that the index definition defines the columns in is important, because in a SELECT query you can use any number of columns starting from the left side of the index column list
- an ordered index can be scanned also in the reverse order, so DESC is only relevant for multi-column indexes

Index on expression (PMOL)

- indexes an expression, e.g. the index: "CREATE INDEX index1 ON table1 ((UPPER(col1)))" can be used in this query: "SELECT * FROM table1 WHERE UPPER(col1) = 'value'"

Covering index (PS)

- extends index leaves to contain values of coli [,...] columns, so the query: "SELECT coli FROM table1 WHERE col1 = value1" will be an index-only scan (it will be very fast)
- each index always automatically includes: (M: PK columns) (S: clustered index key)

Partial index (PSL)

- indexes only rows which satisfy a given condition, e.g. "WHERE col1 IS NOT NULL"
- it is smaller and faster
- only some types of conditions can be used - consult the documentation of a DBMS e.g. SQLite, if the index isn't used by a query
- If its definition includes the UNIQUE keyword, then every entry in the index needs to be unique. This provides a mechanism for enforcing uniqueness across some subset of the rows in a table.

Prefix index (M)

- indexes only the first n characters of a VARCHAR/TEXT column

Index with optimization for the last-page insert contention (OS)

- useful for indexing data such as sequence numbers, where the value of a key increases monotonically
- particularly important in high volume transaction processing systems because:
  - for INSERT: it reduces contention for index blocks,
  - for DELETE: the space used in the index by the deleted row can be reused more easily.
- Oracle: see the "REVERSE" option (**reverse index**)
  - an index reverses the bytes of the key value
  - an index isn't useful for range queries, but range queries are uncommon for artificial values such as sequence numbers
- SQL Server ≥ 2019: see the "OPTIMIZE_FOR_SEQUENTIAL_KEY = ON" option

Index defined on a materialized view (POS) - (S: see "indexed view")

Clustered index (MOS; according to use-the-index-luke.com it should be used only in special situations, it is better to use a "covering index"; unfortunately it is used in MySQL and SQL Server by default)

- it is an index that is integrated with the table physically sorted on a HDD by a column or a list of columns (a table can have only one such index); conversely, a non-clustered index is stored on the HDD separately from the table
- range queries (reading a range of the rows) from it are fast, because DBMS don't have to go first to the index and then to the table, and additionally rows with similar values are close on HDD (reading from HDD is faster)
- writing to a table with a clustered index is slower if there is a need to rearrange the data
- similar to a telephone book, which is sorted by a pair of columns: last name, first name
- M: always created for the PK
  S: by default created for the PK
  O: optional, known as index-organized table, rarely used
  P: not available; to speed up range queries use index-only scans, or manually sort the rows by running: "CLUSTER table1 USING index1"
  L: manually sort the rows, or use WITHOUT ROWID - todo: I don't understand the connection with AUTOINCREMENT - see Clustered Indexes and the WITHOUT ROWID Optimization
- It only makes sense if you often run a range query on this key and the table has no other indexes. So, it doesn't make much sense for a generated primary key, as it is in MySQL or by default in SQL Server.
- The alternative is a non-clustered index which needs an extra IO for each row in the worst case. However it is pretty easy to avoid this overhead by including all needed columns in the non-clustered index (see "covering index"). So, the "index-only scan" is the important concept - not the "clustered index".

Cluster (Oracle, aka. key grouping, rarely used) (todo: is it also in other DBMSs?)

- records of different tables are stored in the same HDD blocks (e.g. records with the same value of the department_id field in the Employee and Department tables), which speeds up joining tables
- see "CREATE CLUSTER ..." in Oracle

Other information

- NULL values are also indexed in a B+ tree index, so the "SELECT * FROM table1 WHERE col1 IS NULL" query can use the index. Oracle is an exception, because it doesn't include rows in a B+ tree index if all indexed columns are NULL.
- By default, table1 (M: isn't) (POS: is) locked during creating and deleting a B+ tree index. If you use the option: (P: "CONCURRENTLY" after "INDEX" keyword) (O: "ONLINE" at the end of the statement) (S: "WITH (ONLINE = ON)" at the end of the statement, available only in Enterprise Edition), then table1 isn't locked, but the operation takes longer.
- syntax of CREATE/DROP INDEX isn't standardized
- only basic information is provided for Oracle and SQL Server
- the syntax for a hash index (possible in Postgres and MySQL) is omitted, because hash indexes are rarely used
- lots of interesting information can be found at use-the-index-luke.com