

Coding and Indexing Strategies for Optimal Performance

Jay Pipes

Community Relations Manager, North America

(jay@mysql.com)

MySQL AB

Agenda

- Index Concepts
- What Makes a Good Index?
- Identifying Good Candidates for Indexing
- Schema Design Considerations
- Index Tips
- Multiple Key Caches
- SQL Coding Techniques and Tips
- Questions

Index Concepts

- Advantages
 - *SPEED* of SELECTs
 - Reporting, OLAP, read-intensive applications
- But...
 - Slows down writes (more work to do...)
 - heavy write applications (OLTP) be careful
 - More disk space used

Index Concepts (cont'd)

- The sort order
 - Enables efficient searching algorithms
- Data Records vs. Index Records
 - Index records “slimmer” than data records
- The layout
 - Hash vs. B-Tree vs. Other
 - Some layouts work well for different types of data access
- The structure
 - Clustered vs. Non-clustered Data/Index Organization

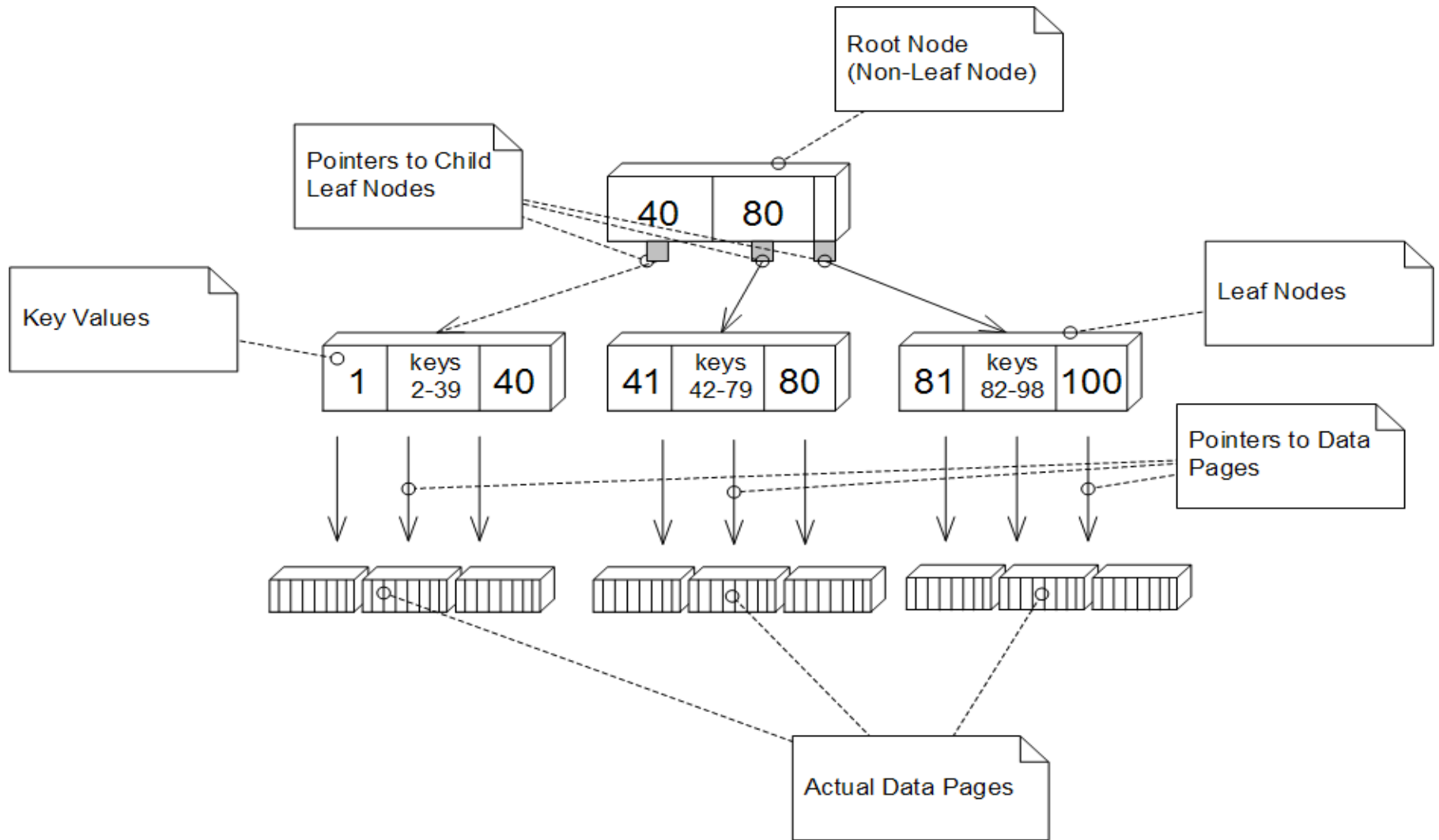
The Scan vs. Seek Choice

- Disk storage dictates choice
 - “Random” access more expensive than sequential access
 - Requires more work than loading a “chunk” of contiguous space and reading through the chunk sequentially
- Threshold
 - Optimizer chooses to scan vs. seek based on:
 - The relative cost of doing a scan vs. a seek (internal)
 - The number of records it estimates will be found by the query.
 - » ~30% of total count of records, optimizer will usually choose to scan vs seek.
 - ANALYZE TABLE

Index Layouts

- Hash Layout
 - Hashing algorithm produces an integer representing the index record value
 - MEMORY and InnoDB (Adaptive). Not MyISAM
 - Very fast for key lookups. Bad for range lookups

B-tree Index



Index Layouts

- B-tree Layout
 - MyISAM
 - Stored in .MYI file (All indexes on table in same file)
 - InnoDB (B+-tree)
 - Stored in data pages (clustered or secondary)
 - Called B+-tree because leaf node also contains pointer to “next” leaf page (this helps in range operations)
 - MEMORY (4.1+)
 - Allows efficient access for ranges of data, or single lookups
 - “Filled” pages to avoid splits
 - Designed to spread key values evenly across the tree
 - » So, index key distribution is important

Clustered vs. Non-clustered

- Describes *whether the data records are stored on disk in a sorted order*
 - MyISAM: Only non-clustered. .MYD=data records, .MYI=index records
 - InnoDB: Only Clustered. *Secondary* indexes built upon the clustering key
- Non-clustered index organization
 - Data not stored on disk in a sorted order, index records are
 - Must do a lookup from the index record to the data record
 - Covering indexes! If index contains all the fields, then no lookup!

Clustered vs. Non-clustered (cont'd)

- Clustered index organization
 - Data is stored on disk in *sorted order*
 - This is why you *must* have a primary key in each InnoDB table
 - So, no lookup from the leaf index page to the data records
 - Why? Because the leaf node **IS** the data page.
 - Consequences?
 - Excellent for primary key access (no lookup required!)
 - Excellent for range requests (no multiple lookups)
 - Easier on updates (no duplication of updates)
 - Panacea?

Clustered (cont'd)

- Disadvantages
 - No more “slim” index pages
 - Because the data resides within the leaf nodes of index, more space in memory needed to search through same amount of records
 - Clustering key appended to *every secondary index record, so....choose clustering key wisely*
- Secondary indexes
 - Similar to MyISAM's non-clustered indexes
 - Built *upon* the clustered data pages
 - Look for covering index opportunities

What Makes a Good Index?

- Selectivity
 - The relative *uniqueness* of the index values to each other
 - $S(I) = d/n$
 - Unique index always has a selectivity of 1
 - The higher the better? Depends...
 - Multiple columns in an index = multiple levels of selectivity
 - So how to determine selectivity? ...

Determining Selectivity – The Hard Way

for each table in your schema...

```
SELECT COUNT(DISTINCT field) FROM table;
```

```
SELECT COUNT(*) FROM table;
```

or

```
SHOW INDEX FROM table;
```

Determining Selectivity – The Easy Way w/ MySQL 5!

```

SELECT
  t.TABLE_SCHEMA
  , t.TABLE_NAME
  , s.INDEX_NAME
  , s.COLUMN_NAME
  , s.SEQ_IN_INDEX
  , (
    SELECT MAX(SEQ_IN_INDEX)
    FROM INFORMATION_SCHEMA.STATISTICS s2
    WHERE s.TABLE_SCHEMA = s2.TABLE_SCHEMA
    AND s.TABLE_NAME = s2.TABLE_NAME
    AND s.INDEX_NAME = s2.INDEX_NAME
  ) AS "COLS_IN_INDEX"
  , s.CARDINALITY AS "CARD"
  , t.TABLE_ROWS AS "ROWS"
  , ROUND(((s.CARDINALITY / IFNULL(t.TABLE_ROWS, 0.01)) * 100), 2) AS "SEL %"
FROM INFORMATION_SCHEMA.STATISTICS s
INNER JOIN INFORMATION_SCHEMA.TABLES t
  ON s.TABLE_SCHEMA = t.TABLE_SCHEMA
  AND s.TABLE_NAME = t.TABLE_NAME
WHERE t.TABLE_SCHEMA != 'mysql'
AND t.TABLE_ROWS > 10
AND s.CARDINALITY IS NOT NULL
AND (s.CARDINALITY / IFNULL(t.TABLE_ROWS, 0.01)) < 1.00
ORDER BY t.TABLE_SCHEMA, t.TABLE_NAME, s.INDEX_NAME, "SEL %" \G

```

Identifying Good Field Candidates for Indexes

- WHERE clauses
 - What are you filtering by?
 - Good distribution and selectivity in field values
 - Don't index “is_active” fields! (bad selectivity)
 - Caution on indexing “status” fields (bad distribution)
- ON conditions
 - Foreign keys...
 - Usually the primary key is indexed... but **MAKE SURE!**
 - Performance consequences if you don't index ON condition fields used in an eq_ref or ref_or_null access
 - Look here for excellent covering index opportunities!

Identifying Good Field Candidates for Indexes (cont'd)

- GROUP BY clauses
 - Pay attention here! Field order is important.
 - VERY important to note how you are grouping.
 - Very common Web 2.0 example:

```
CREATE TABLE Tags (
tag_id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY
, tag_text VARCHAR(100) NOT NULL
) ENGINE=MyISAM;
```

```
CREATE TABLE ProductTags (
product_id INT UNSIGNED NOT NULL
, tag_id INT UNSIGNED NOT NULL
, PRIMARY KEY pk_ProductTags (product_id, tag_id)
) ENGINE=MyISAM;
```

Identifying Good Field Candidates for Indexes (cont'd)

- What happens on a SELECT grouped by the project?

```
SELECT product_id, COUNT(*) as tags  
FROM ProductTags  
GROUP BY product_id;
```

- What happens on a SELECT grouped by the tag?

```
SELECT tag_id, COUNT(*) as projects  
FROM ProductTags  
GROUP BY tag_id;
```

- Solution: Add index on (tag_id) or covering on (tag_id, product_id)

Identifying Good Field Candidates for Indexes (cont'd)

- Identifying redundant indexes where one index is entirely contained within another. Example:

```
CREATE TABLE Products (  
  product_id INT UNSIGNED NOT NULL PRIMARY KEY  
  , vendor_id SMALLINT UNSIGNED NOT NULL  
  , supplier_id SMALLINT UNSIGNED NOT NULL  
  , product_type SMALLINT UNSIGNED NOT NULL  
  , INDEX ix_TypeVendor (product_type, vendor_id)  
  , INDEX ix_Redundant (product_type) # Entirely contained by above!  
  ) ENGINE=MyISAM;
```

Schema Design Considerations

- Your schema design will affect performance!
- If a natural key exists, *don't add a surrogate key!*
 - Especially true on InnoDB tables because of benefit of clustered organization

Schema Design Considerations (cont'd)

- Keep data types as small as possible for what you need
 - Don't use BIGINT unless you have to! Waste of space.
 - Especially true on indexed columns
 - Fundamental principle:

The smaller your data types, the more records will fit into the index blocks. The more records fit in each block, the fewer reads are needed to find your records.

Schema Design Considerations (cont'd)

- “Horizontal Splitting” of tables into multiple tables
 - When you have many NULLable fields or seldom-queried fields, split into main and “extra” tables (One-to-One relationship)
 - Allows more “main” records to be cached
- “Vertical Splitting”
 - MERGE tables
 - Partitioning in 5.1

Schema Design Considerations (cont'd)

- Choosing a clustering key
 - Remember, clustered index good for both range and single-key lookups, *but* clustering key will be appended to every secondary index record
 - Recommendation: Use `AUTO_INCREMENT`ing clustering key unless special situations
 - Read-only, OLAP, reporting: often useful cluster on date/dimension because of range considerations

Schema Design Considerations (cont'd)

- InnoDB, important:
 - Don't use SELECT COUNT(*) queries.
 - Use summary tables which maintain counters for the stuff for which you would normally be using COUNT(*)
- Use summary tables for reporting solutions
 - Use MyISAM unless there's a specific reason no to
 - Replaces multiple SELECT ... GROUP BY queries with single calls to SELECT ... FROM *summaryTable*
 - Less/no tmp tables, filesort issues
 - No lock contention (added bonus: sticks in the qcache)

Index Tips

- Tackle the big stuff first!
 - Don't waste time optimizing from 5 ms to 1ms!
 - Use the Slow Query Log to identify troublespots
 - Profile your queries with EXPLAIN, MyTop, etc
- Look for covering index opportunities
 - Key Mapping tables (Many-to-Many relationships)
 - Remember to consider the *order* of fields in index
- Highly selective fields will perform best
 - Put less selective fields on the LEFT side of multi-column indexes. Helps with GROUP BY on the less-selective fields
 - By same token, get rid of indexes on fields with no selectivity

Index Tips (cont'd)

- Make sure key relation fields are the same data type
- If you change data types in your main table, ensure child tables change as well!
- Only index the portion of the field that is useful
- For character field indexes, there's little need to index the entire width of the field
 - `CREATE INDEX ix_firstname ON (firstname(10));`
- Saves index space (less space = more records = faster search)

Multiple Key Caches (MyISAM only)

- Multiple Index Caches (MyISAM only)
 - If you know beforehand that certain tables are used more often than others, assign them to their own cache
 - Setup the separate caches
 - SET GLOBAL hot_cache.key_buffer_size=128*1024; *or*
 - hot_cache.key_buffer_size = 128K (recommended)
 - Assign tables to different caches
 - CACHE INDEX zip_lookup, location_lookup TO hot_cache;
 - CACHE INDEX cart_contents, cart_order TO cold_cache;

Multiple Key Caches (cont'd)

- Destroyed upon server restart
- `init_file=/path/to/data-directory/mysqld_init.sql`
 - In `mysqld_init.sql`:
 - `CACHE INDEX zip_lookup, country_lookup IN hot_cache;`
- Noticing a lot of swapping in your caches?
 - Mid-Point Insertion Strategy
 - Caused by index scans
 - You will see `key_blocks_unused = 0`
 - “Pin” important index blocks into memory with `key_cache_division_limit < 100`:
 - `SET GLOBAL default.key_cache_division_limit = 70;`

Multiple Key Caches (cont'd)

- Preload your indexes for maximum efficiency
 - Why? Sequential load vs. Random load
 - `LOAD INDEX INTO CACHE zip_lookup;`
 - Use `IGNORE LEAVES` clause to only sequentially load non-leaf nodes

Symlinking Indexes to Separate Drives (MyISAM)

- Since you don't care about redundancy...
 - Place your index files on RAID0 array (ex: /data0)
 - While mysqld is running...
 - `CREATE TABLE ... INDEX DIRECTORY=/data0/db1/;`
 - If mysqld is not running...
 - Move the index file to /data0/db1/tbl1.MYI
 - `#> ln -s /path/to/datadir/db1/tbl1.MYI /data0/db1/tbl1.MYI`

SQL Coding Techniques for Optimal Performance

- Keys to fast, reliable, and error-free SQL coding
 - Specificity
 - Consistency
 - Joins!
 - Keep It Simple
 - “Chunky” Statements
 - Isolate Your Indexed Fields
 - Steer Clear of Non-deterministic functions
 - Use SQL_NO_CACHE correctly

Specificity

- ANSI style vs. Theta style (hint: use ANSI)
- ANSI example

```
SELECT coi.order_id, pr.product_id, p.name, p.description
FROM CustomerOrderItem coi
  INNER JOIN Product p
    ON coi.product_id = p.product_id
WHERE coi.order_d = 84462;
```

- Theta style example:

```
SELECT coi.order_id, pr.product_id, p.name, p.description
FROM CustomerOrderItem coi, Product p
WHERE coi.product_id = p.product_id
AND coi.order_d = 84462;
```

Consistency

- Don't mix and match styles
- Especially important in team environments!
 - Develop a Coding Style and Format Guideline
 - Sounds over the top, but it will save you many a headache
- Don't mix and match LEFT and RIGHT join
 - There is no difference between them (just what side of the ON condition you put the tables)
 - Use LEFT, only LEFT. Clear and consistent
- Leads to code that is:
 - Easier to read
 - Easier to maintain

Learn to Use Joins !!!!!!!

- Perhaps the most important key of all
- A good SQL coder can reduce many piles of “spaghetti subqueries” to a single join
- Joins are much more *flexible*
 - Try accomplishing an outer join with a correlated subquery in a WHERE clause. Hint: you can't.
- Correlated subqueries are EVIL.
 - Optimizer has a tough time with them
 - Almost always can be written more efficiently as a standard join or a derived table.
 - Jay, what's a derived table?

Join vs Correlated Subqueries in SELECT

- Correlated Subquery (1 lookup access *per product_id!*)

```
SELECT pr.product_id, p.name
, (SELECT MAX(coi.price)
  FROM CustomerOrderItem
  WHERE product_id = p.product_id) as max_price_sold
FROM Product p;
```

- Join with GROUP BY (1 query on CustomerOrderItem, *then join*)

```
SELECT pr.product_id, p.name, MAX(price) as max_price_sold
FROM Product p
INNER JOIN CustomerOrderItem coi
  ON p.product_id = coi.product_id
GROUP BY product_id;
```

Derived Table vs Correlated Subqueries in WHERE

// Correlated subquery

```
SELECT
c.company
, o.*
FROM Customers c
  INNER JOIN Orders o
    ON c.customer_id = o.customer_id
WHERE order_date = (
SELECT MAX(order_date)
FROM Orders
WHERE customer = o.customer
)
GROUP BY c.company;
```

// Derived Table

```
SELECT
c.company
, o.*
FROM Customers c
  INNER JOIN (
  SELECT
  customer_id
  , MAX(order_date)
    AS max_order
  FROM Orders
  GROUP BY customer_id
) AS m
  ON c.customer_id = m.customer_id
  INNER JOIN Orders o
    ON c.customer_id = o.customer_id
    AND o.order_date = m.max_order
GROUP BY c.company;
```

Keep It Simple with “Chunky” Coding

- Break complex questions into smallest parts
 - Leads to you thinking in terms of derived tables, versus correlated subqueries
- Avoid making stored procedures that try to accomplish everything all at once.
 - Break procedures into small, re-usable “components”
 - Same with views.
- Think in terms of *sets*, not cursors
- A complex SELECT statements is simply the joining together of multiple sets of data. If you've got a tough SELECT statement, then break it down by identifying the unique sets of data you're dealing with.

Isolate Your Indexed Fields

- Don't have functions operating on indexed fields.
 - Prevents the index from being used
 - Instead of: `WHERE TO_DAYS(order_created) – TO_DAYS(NOW()) <= 7`
 - USE: `WHERE order_created >= DATE_SUB(NOW(), INTERVAL 7 DAY)`
- If you can't re-arrange the equation, consider a calculated field
 - Example: trying to find all emails from “aol.com”
 - Not good: `WHERE email_address LIKE '%aol.com'`;
 - Better to create a field in table “rv_email_address”
 - Have a trigger which automatically inserts correct reversed field value into the rv_email_address field
 - Then: `WHERE rv_email_address LIKE CONCAT(REVERSE('aol.com'),'')`

Non-Deterministic Functions and SQL_NO_CACHE

- Non-deterministic functions
 - Functions which will not return the same value, given the same parameters
 - Examples: NOW(), CURRENT_USER(), CURRENT_DATE()
 - Steer clear because prevents the query_cache from caching the results of the query
- On heavily updated tables, use SELECT SQL_NO_CACHE
 - No use wasting the server's energy putting the result into the query_cache if it will just be invalidated moments later
 - Example: session tables, volatile statistic tables
 - Use InnoDB for these types of tables
 - Row-level locking best for high-write tables

SQL Coding Techniques for Optimal Performance (cont'd)

- Avoid the terrible twins:
 - NOT IN and <>
- Don't try to out-think the optimizer. Join Hints are almost always a bad idea.
- Use stored procedures!
 - PeterZ's initial studies show performance improvement of 22% vs. MySQL 4.1 same exact queries w/o stored procedures
 - Better encapsulation of code and logic