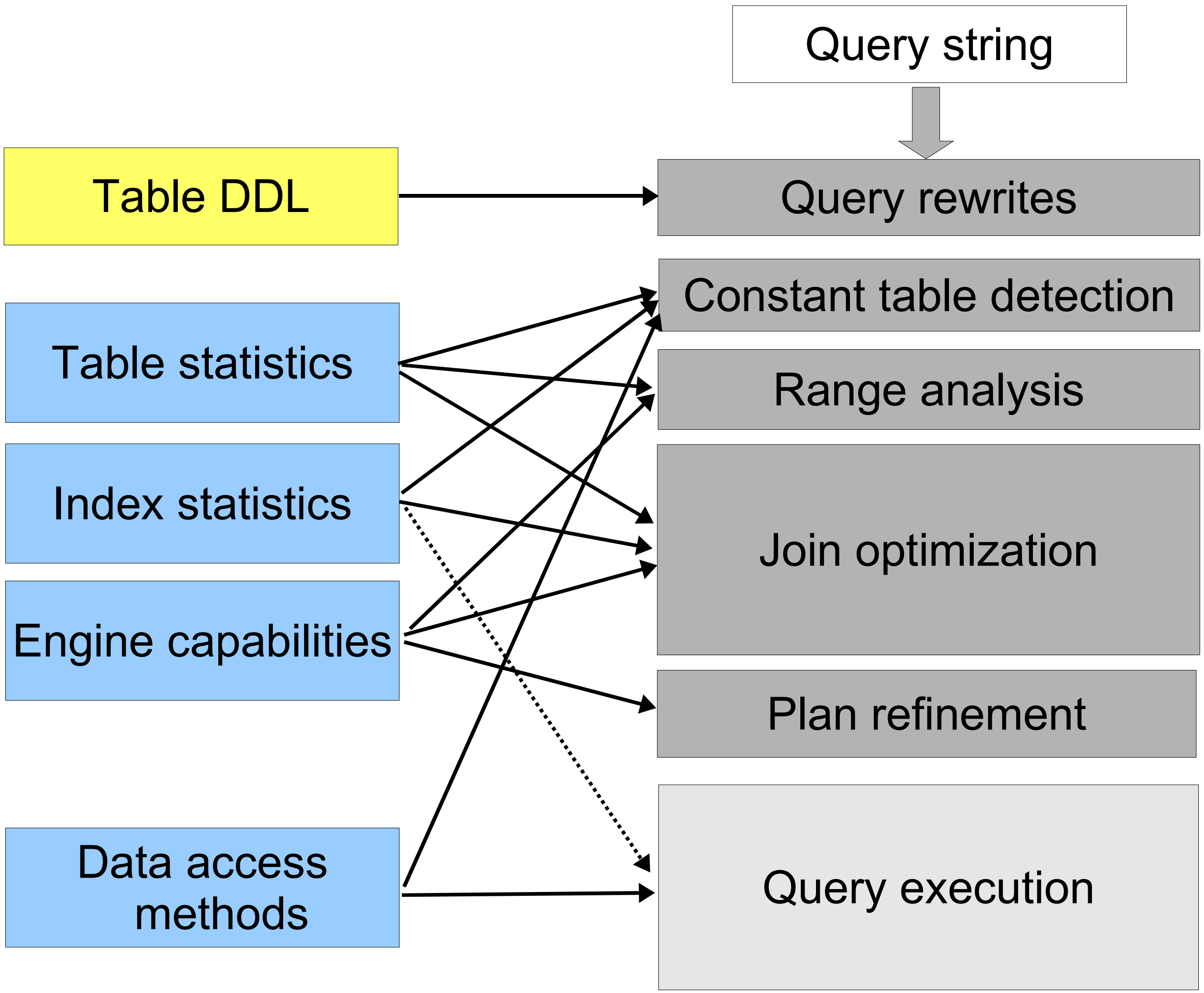
Interaction Between Optimizer and Storage Engine

MySQL University Session

Sergey Petrunia <sergefp@mysql.com>

Dec 06 2007

Query optimization and execution scheme



Input data for the optimizer: general idea

The optimizer needs to know:

- Storage engine capabilities
  - to see if certain execution strategies can be used
    - e.g. if index scans produce data in order
- Costs of doing table/index/range/etc scans
- Numbers of matching records
  - Number of records that will be produced when using some access method
  - Number of records that will match certain parts of the WHERE
- Certain table properties for heuristics
  - e.g. whether the primary index is a clustered index

Input data for the optimizer: concrete list

- Table DDL information
  - Column types, charsets, nullability
  - Indexes
    - But not PK/FK relationships
  - Partitioning info
- Table engine information
  - Table flags
  - Index flags
- Statistics
  - Table statistics
  - Index statistics (cardinalities)
  - records_in_range estimates

SE<->Optimizer interface walkthrough

1. Per-table statistics
2. Full table scans
3. Index-based access functions
4. Ref access
5. Full index scan
6. Range scan
7. Multi Range Read interface
8. rnd_pos() and its usage
9. index_merge scan
10. Table condition pushdown
11. Index condition pushdown

Per-table statistics: handler->stats

- **handler->stats.records**
  - This is an estimate of how many records are in the table
  - Filled by handler->info(HA_STATUS_VARIABLE)
  - h->ha_table_flags() & HA_STATS_RECORDS_IS_EXACT: values of 1 or 0 mean table will have 1 or 0 records
    - Important special case, used to detect const tables
- **handler->estimate_rows_upper_bound()**
  - Upper bound of #records in the table. ATM used by filesort()
- **handler->stats.mean_rec_length**

  to be used to estimate space for sorting/subquery materialization/etc
- **Optionally handler->stats.data/index_file_length**

  used by default implementations of access cost functions.

Full table scan

- Execution:

```
handler->rnd_init() = 0
handler->extra(HA_EXTRA_CACHE) = 0
handler->rnd_next() = 0
...
handler->rnd_next() = 0
handler->rnd_next() = HA_ERR_END_OF_FILE
handler->rnd_end() = 0
handler->extra(HA_EXTRA_NO_CACHE) = 0
```

- Cost function

```
double handler::scan_time()
{ return ulonglong2double(stats.data_file_length) / IO_SIZE + 2; }
```

Index-based access functions

- SE API has an set of functions: basic navigation, range read, multi-range-read

multi_range_read_init()			multi_range_read_info()
multi_range_read_next()			multi_range_read_info_const()
	read_range_first()	read_range_next()	records_in_range()
			index_only_read_time()
			read_time()
		index_next_same()	
index_first()	index_read_map()	index_next()	
index_last()	index_read()	index_prev()	

Access methods that use index-based functions

- **index** - index_first/last, index_next/prev
- **range** - multi_range_read_XXX()
  - use read_range_first/next() in default implementation
    - use index_first/read/next/next_same in default implementation
- **index_merge**
  - See range.
- **ref**
  - Currently index_read/index_next_same
  - multi_range_read_XX in MySQL 6.0
- **'Using index for group-by'**
  - index_read/index_next/index_prev
- One-lookup table read, one-lookup MIN/MAX resolution:
  - index_read

Non-batched ref access

- ref access = index lookups over [prefix] equality

```
keypart1=e1 AND keypart2=e2 AND ... AND keypart_k=eK
```

- Execution:

```
h->index_init(index_no, sorted=FALSE) = 0
...
// Lookup start
h->index_read_map('lookup-key', HA_READ_KEY_EXACT) = 0
h->index_next_same('lookup-key') = 0
h->index_next_same('lookup-key') = 0
...
h->index_next_same('lookup-key') = HA_ERR_END_OF_FILE
// Lookup end
...
handler->index_end()=0
```

- Variants:
  - eq_ref doesn't call index_next_same()
  - ref_or_null makes second lookup with NULL key

Non-batched ref access, cost calculations

- Number of records we get in one lookup
  – handler->info(HA_STATUS_CONST) fills handler->table->key_info[*all_keys*].rec_per_key[*all_keyparts*]
  – this is E(#matching records for one lookup)
  – Cardinality in SHOW KEYS is #records/rec_per_key
  – Value of 0 means “unknown”
  – NULLs problem:
    - Sometimes NULLs should be ignored, sometimes treated as equal values (see BUG#9622)

- One index lookup cost
  – index_only_read_time(1 range, n_rows), or
  – rows2double(n_rows)
    - not read_time(1 range, n_rows) as one could think

rec_per_key: details and the NULLs problem

- rec_per_key[k]= { E (#rows) | keypart1 = c1 AND
  keypart2 = c2 AND
  ...
  keypartK = cK}

- The optimizer assumes that *every* index lookup will find rec_per_key[#n_keyparts_used] matches
  - Even if there is a PK/FK relationship which shows that that is not true
- The optimizer uses rec_per_key value even if ref access use "keypart_i IS NULL"
  - This can give wrong estimates because NULLs are "special" values (there are often more NULLs than any other value)
  - MyISAM has several statistics collection methods (see **myisam_stats_method**) but they all have different flaws
    - We're working on some scheme that will store/use information about numbers of NULLs

Full index scan

- Forward: just like lookup but starts with index_first():

```
handler->extra(HA_EXTRA_KEYREAD) = 0
handler->index_init(index1, sorted=TRUE) = 0
handler->index_first() = 0
handler->index_next() = 0
...
handler->index_next() = HA_ERR_END_OF_FILE
handler->extra(HA_EXTRA_NO_KEYREAD) = 0
handler->index_end() = 0
```

- Backwards scan:
  - Same as above but uses index_last/index_prev
- Cost:
  - index_only_read_time(1 range, #rows_in_table)

Non-batched range scan

- Execution:

  handler->index_init(index1, sorted=...) = 0

  ...

  // range x start

  handler->read_range_first(left_endp, right_endp, sorted=...) = 0

  handler->read_range_next() = 0

  ...

  handler->read_range_next() = HA_ERR_END_OF_FILE

  // range x end

  ...

  handler->index_end() = 0

- Also
  – HA_EXTRA_KEYREAD may be in effect
  – Ranges are disjoint and ordered
  – read_range_XXX() don't allow to scan backwards. Reverse range scans use index_prev() calls

Non-batched range scan, optimization

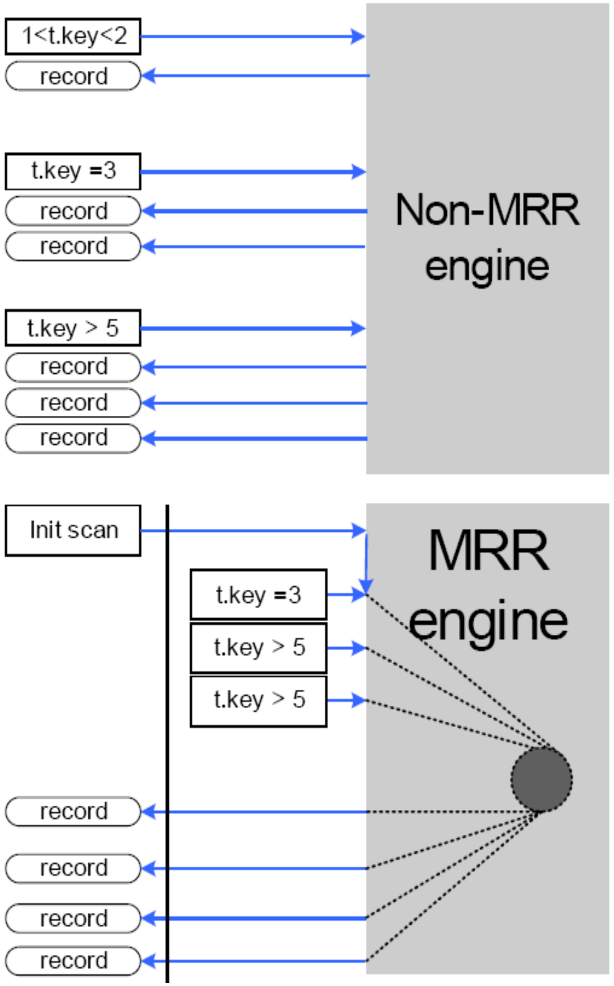
```
// find out how many records in all ranges
// also check if engine is able to scan such ranges
for each range {
  rows= h->records_in_range(index1,  left_endp, right_endp) = 0
  if (rows == HA_POS_ERROR)
    break;
  total_rows += rows;
}
//get the cost
if (index_only)
  cost= h->index_only_read_time(keyno, total_rows);
else
  cost= h->read_time(keyno, n_ranges, total_rows);
```

records_in_range() estimate properties

- Returning 0 from **records_in_range()** is interpreted as a a statement that there will be no matching records.
- The optimizer assumes that **records_in_range()** estimates are rather precise
  - Values obtained from rec_per_key are adjusted if they are in contradiction with records_in_range() call results
  - Even if range scan is not used, records_in_range() value is used to get an estimated number of records that will match the table's condition.
- The optimizer tries (and will try harder) to avoid making too many records_in_range() calls.

Multi Range Read interface

- Both optimization and access operate on batches of ranges
- Used for range scans now
- Will be used to batch ref scans in MySQL 6.0 (WL#2771)
- Default implementation converts calls to range-based functions
- NDB has custom implementation now
- MyISAM/InnoDB will have custom implementation (DS-MRR) in 6.0



Multi Range Read interface usage pattern

```
// Optimization:
// when ranges are not known in advance:
h->multi_range_read_info() = #rows and other info
// when ranges are known in advance:
h->multi_range_read_info_const() = #rows and other info

// Execution
h->multi_range_read_init(range_sequence) = 0
h->multi_range_read_next() = 0
h->multi_range_read_next() = 0
...
h->multi_range_read_next() = HA_ERR_END_OF_FILE
```

position() and rnd_pos() calls

- Used to remember record rowids and get records later
  - position() is used to save the rowid
  - rnd_pos() gets a record from rowid
- Used by
  - UPDATE/DELETE code when updating several tables or updating the index we're scanning
  - index_merge code
  - filesort() over tables with blobs
- No cost methods atm
  - index_merge uses its own calculations
  - Other users don't do cost-based choice
- Note: it's ok to return HA_ERR_RECORD_DELETED from rnd_pos() call.

How index_merge uses handler interface (1)

- Sort-union index_merge execution:
  - range scan on 1st merged index
  - range scan on 2nd merged index
  - ...
  - range scan on Nth merged index
  - rnd_pos() scan
    - Sequence of rnd_pos() calls, all rowids are distinct and are passed in order.

- Sort-union index_merge soptimization
  - Range access estimate calls for each of the indexes
  - Cost of rnd_pos() scan is calculated at the SQL layer
    - It is assumed to be faster than just n rnd_pos() calls because rowids will be passed in their order.

How index_merge uses handler interface (2)

union/intersection execution

- Index scans must have ROR (RowidOrderedRetrieval) property: a scan on

  keypart1=const1 AND ... AND keypartN=constN

  must return records in rowid order
  - where handler->cmp_ref() is the rowid ordering function
- handler->primary_key_is_clustered() => any primary key scan is a ROR scan.
- For non-ROR indexes:

  index_flags(idx,0, TRUE) & HA_KEY_SCAN_NOT_ROR

- Optimization
  - Cost calculations are done at SQL layer
  - SQL layer may make records_in_range() calls for ranges it is not going to scan.

Table condition pushdown

- One handler function:

  Item *handler::cond_push(Item* cond)

- Is useful for engines that have “smart” storage but limited bandwidth to it
- ATM condition pushdown is implemented only by NDB
  - Should be implemented by federated but isn't
  - Don't re-use NDB implementation, approach at make_cond_for_table/index() is more powerful
- API is not stable
  - And not compatible across versions

Index condition pushdown

- One handler function:

  Item *handler::idx_cond_push(uint keyno, Item* cond)

- Is useful for storage engines that pay extra for reading the complete table record
- Works with any index-based scan
  - SQL layer won't call it for 'index' access
- In MySQL 6.0 is implemented for MyISAM and InnoDB
- Same API notes as in table condition pushdown.

Challenges in SE<->optimizer interface

- The interface is a product of step-by-step improvements, not design
- Some optimizations are relevant to one kind of engine but not the others
- Problem with cost function encapsulation:
  - On one hand, need to ask storage engine about everything, without making any assumptions
  - On the other hand, cannot run optimization on opaque functions – minimizing cost requires knowledge of the form of the function

Future plans

In no particular order:

- Better rec_per_key estimates for various cases with NULLs
  – e.g. a ref scan on “keypart1=c1 AND keypart2 IS NULL”
- Cleanup in the MRR interface
- Make MRR interface support semi- and outer joins
  – One match only mode
  – Return distinct rows only mode
- Write a plugin that will check if statistics provided by the engine were any good.

The end

Thanks for your attention