

How MySQL handles ORDER BY, GROUP BY, and DISTINCT

Sergey Petrunia, sergefp@mysql.com

MySQL University Session

November 1, 2007

Handling ORDER BY

- Available means to produce ordered streams:
 - Use an ordered index
 - **range** access
 - not with MyISAM/InnoDB's DS-MRR
 - not with Falcon
 - Has extra (invisible) cost with NDB
 - **ref** access (but not ref-or-null)
 - results of `ref(t.keypart1=const)` are ordered by `t.keypart2, t.keypart3, ...`
 - **index** access
 - Use **filesort**

Executing join and producing ordered stream

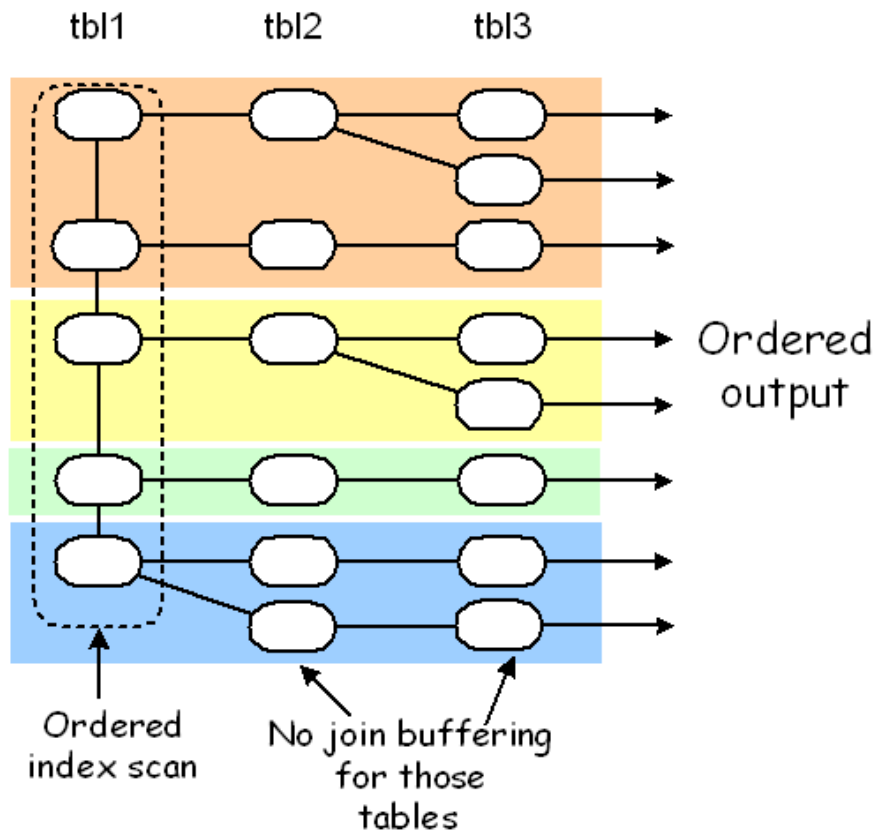
There are three ways to produce ordered join output

Method	EXPLAIN shows
Use an ordered index	Nothing particular
Use filesort() on 1st non-constant table	“Using filesort” in the first row
Put join result into a temporary table and use filesort() on it	“Using temporary; Using filesort” in the first row

EXPLAIN is a bit counterintuitive:

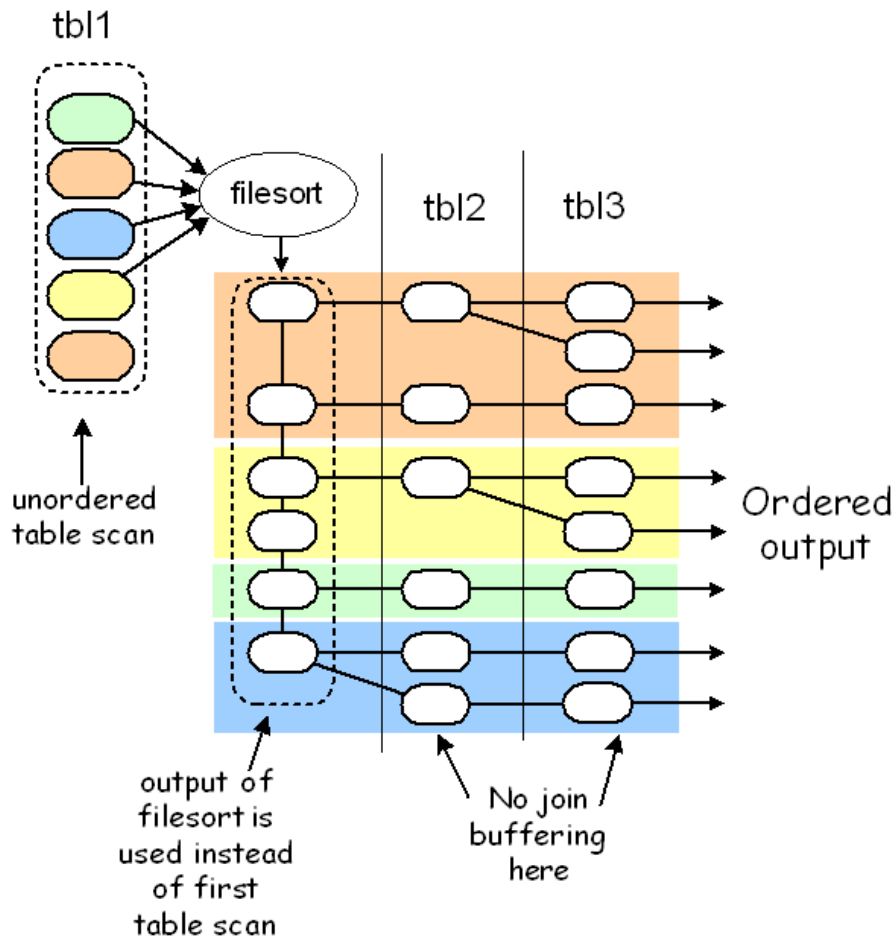
id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	t2	range	a	a	5	NULL	10	Using where; Using temporary; Using filesort
1	SIMPLE	t2a	ref	a	a	5	t2.b	1	Using where

Using index to produce ordered join result



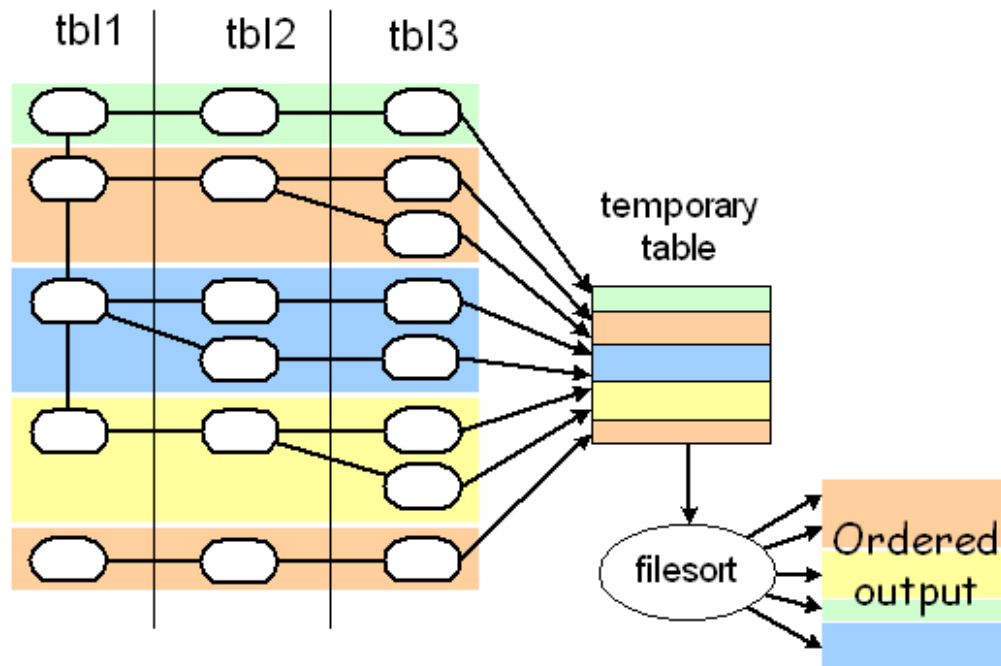
- ORDER BY must use columns from one index
- DESC is ok if it is present for all columns
- Equality propagation:
 - “a=b AND b=const” is detected
 - “WHERE x=t.key ORDER BY x” is not
- Cannot use join buffering
 - Use of matching join order disables use of join buffering.

Use filesort on first non-const table



- Same as previous method, but use filesort instead of an the index
 - All same properties
- Condition on 1st table is checked before filesort
- But LIMIT can't be “pushed down”
- In the code: see `JOIN::simple_order`, `simple_group`

Use filesort for the entire join output



- This is the catch-all method
 - Places no limit on join order, use of join buffering, etc
- Works only for *all* tables (can't filesort {t1,t2})
- LIMIT is not pushed down to before filesort

A more detailed look at filesort

- filesort() operation
 - Do one in-memory qsort if we have enough space
 - Otherwise sort in chunks and then merge them until we get one sorted sequence.
- filesort() modes
 - Sort tuples that contain all required columns from the source table
 - Sort <sort_key, rowid> pairs, return a sequence of source table's rowids
 - One will need to do a separate pass where the rows are read in sort (not disk) order

EXPLAIN does not tell which mode is used, need to use some indirect evidence like Handler_XXX counters.

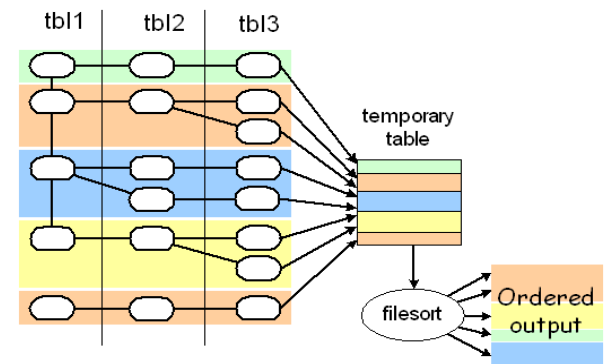
ORDER BY and the join optimizer

A proper approach:

- Have cost of sorting
- Let the optimizer take it into account:

$$\text{query_cost} = \text{cost}(\text{join_execution}) + \text{need_sort? cost_of_sort} : 0$$

- But this is not implemented yet
- Implementation challenges
 - Don't know condition selectivity ->
 - Will need to do smart choice whether to use join buffering

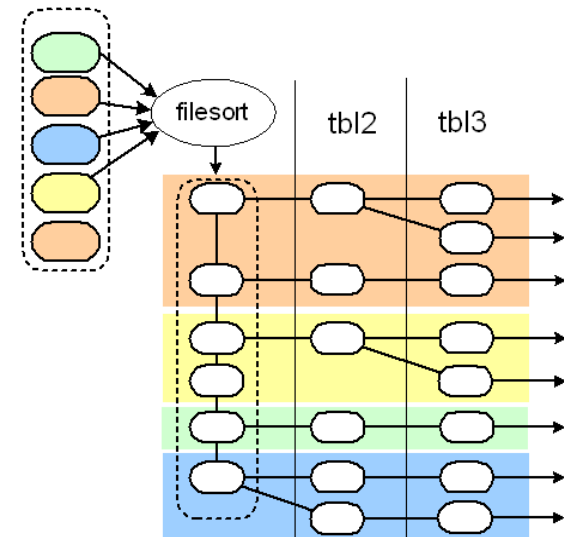


- Solution scheme:
 - Produce best join order ignoring the need to sort
 - Produce best join order that doesn't require sorting
 - Compare those three

ORDER BY and the join optimizer (2)

Currently we have:

- No formula to get cost of sorting
- Join order choice doesn't properly account for cost of sorting
 - Makes an trivial attempt, grep for “have to make a temp table” in `best_extension_by_limited_search()`
 - this can misfire, see `BUG#XXXX`
- Once the join order is fixed, we consider changing access method for 1st table to produce required ordering. The choice is cost-based



Handling GROUP BY

General approach to processing GROUP BY

```
reset all aggregate functions;
for each join record combination (*)
{
    group= {find out the group this record combination falls into}; (**)
    update aggregate functions;
}

for each group
{
    get aggregate function values;
    if (HAVING clause is satisfied)
        pass the group row for further processing;
}
```

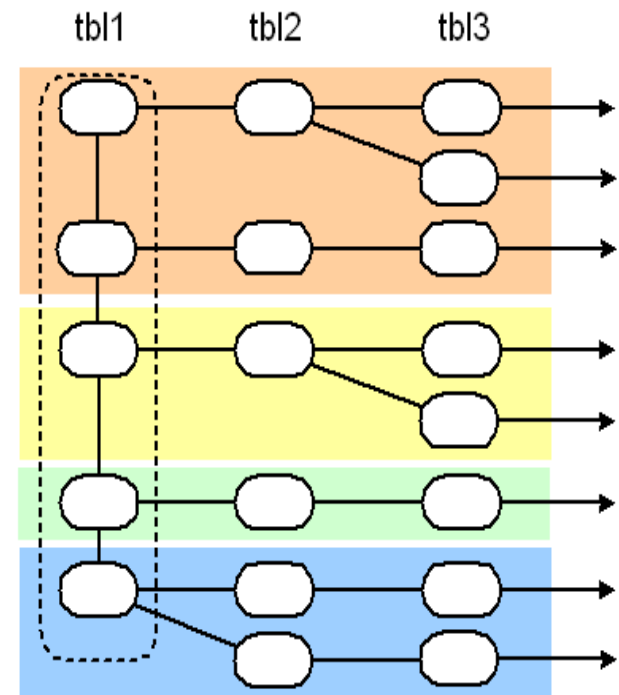
Available GROUP BY strategies

There are three

- **Ordered Index Scan**
execute the join in a way that record combinations are grouped together
- **Loose Index Scan**
handles a special case when there is only one table we need only one row for each group
- **Groups table**
Execute join so we get groups intermixed and use temp. table to find the group.

Handling GROUP BY with ordered index scan

- Groups are processed one after another
- ORDER BY group_by_list is guaranteed for free
- Codewise:
 - Join result is captured with `end_send_group()`
 - Still uses some parts of temptable-based solution (e.g. `join->tmp_table_param`)
 - Because it needs two buffers
 - Current join record combination
 - GROUP values for group being examined



GROUP BY / Loose Index Scan

```
SELECT Country, max(Population)
FROM City
WHERE Name = 'Sorrento'
GROUP BY Country;
```

v. 5.0

1. Find group prefix: 'USA'
2. Compose search key: <'USA', 'Sorrento'>
3. Jump to first key with: <'USA', 'Sorrento'>

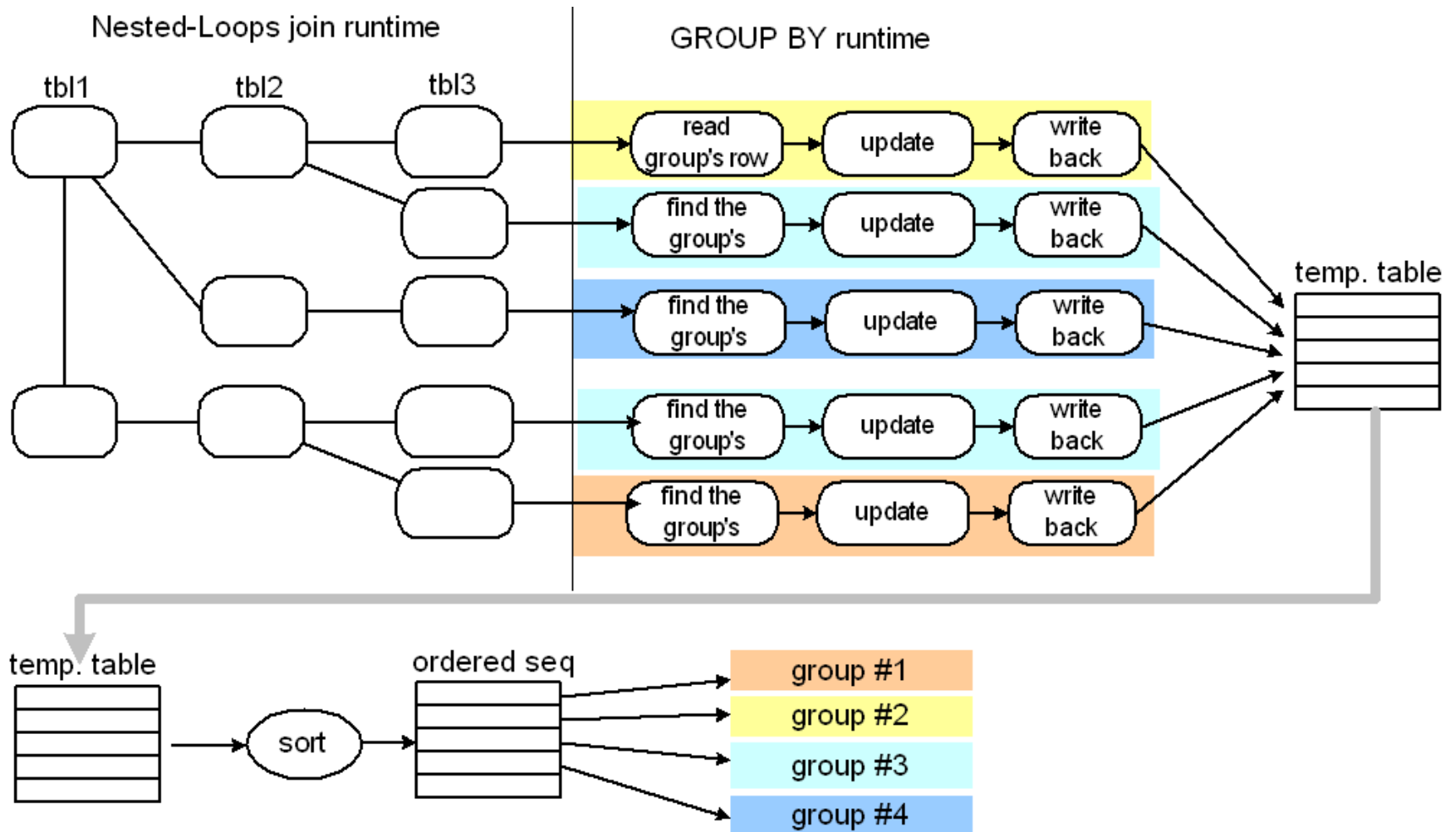
=> **First** row contains MIN value
=> **Last** row contains MAX value

Country	Name	Population
...
URY
USA	Abilene	...
...
USA	Rockford	150115
USA	Sorrento	290
USA	Sorrento	765
USA	Sorrento	1260
UZB
...

GROUP BY / Loose Index Scan (2)

- Limited applicability
 - Join must have only one table
 - Only MIN(col)/MAX(col) functions are allowed
 - The index must cover
 - GROUP BY columns
 - Several equality columns (optional)
 - MIN/MAX column(s)
- Low-cardinality indexes are good (contrary to other access)
- Codewise
 - It's an access method
 - Uses QUICK_GROUP_MIN_MAX_SELECT
 - Produces grouped result right away, doesn't need to capture join output

Handling GROUP BY with groups table



Handling GROUP BY with groups table (2)

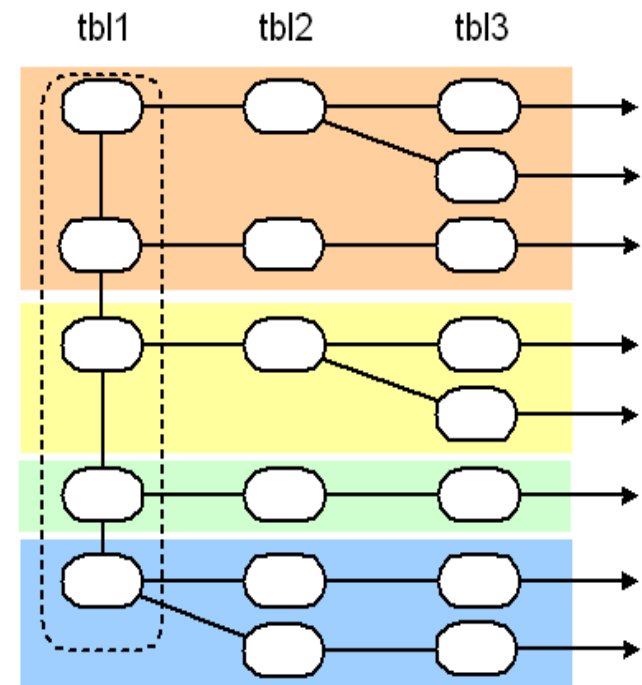
- Groups table has
 - GROUP BY columns
 - with unique index
 - Aggregate function columns
 - This method handles only $O(1)$ data per group aggregates
- Group table properties
 - HEAP table with HASH key by default
 - Auto-converted to MyISAM when full
 - MyISAM table when HEAP cannot be used
 - Regular BTREE key by default
 - Unique constraint (key on hash value) otherwise
- Join results captured by `end_update()`, which
 - Finds the group in the temp table
 - Updates the group values

GROUP BY special cases

- Constants are removed from GROUP list
- GROUP BY all_unique_key_columns is detected and removed
- SELECT MIN(key), MAX(key) without GROUP BY can/is resolved with one index lookup in certain cases
- We don't take advantage of HASH indexes for GROUP BY, although we could do it when using range access over HASH indexes (we don't get ordering but get grouping)

Handling DISTINCT aggregates

- `aggregate_func(DISTINCT ...)` needs to store $O(\#groupsize)$ data per group
- Which can't be done efficiently with group table
- Therefore, we do filesort pass first so we enumerate groups one after another
- DISTINCT group functions accumulate data in UNIQUE objects
(i.e. BTREEs which can be flushed to disk and then merged to get an ordered distinct sequence)



Handling DISTINCT aggregates (2)

Loose Index Scan can be used for DISTINCT aggregates:

```
SELECT COUNT(DISTINCT t.key) FROM t1;
```

```
SELECT COUNT(DISTINCT key1part1)
FROM t1 GROUP BY key1part1, key1part2;
```

- Can handle GROUP BY iff aggregate's list is a prefix of it
- Have limitations on select list and WHERE clause
 - similar to those of GROUP BY's LooseIndexScan
- We've started to implement it, see WL#3220
 - Can provide speedups for low-cardinality indexes
 - Slower than index scan for high cardinality indexes
 - Need cost model to decide when to use LooseIndexScan
 - Need storage engines to support efficient “jump forward” function

Handling SELECT DISTINCT

- The task is similar to GROUP BY
 - Need to detect groups and pick one record from every groups
 - Therefore handled by the same code
 - May be converted to GROUP BY's internal data structure
- As with GROUP BY, trivial cases are detected and removed
 - Constants in select list
 - GROUP BY and DISTINCT on the same columns w/o aggregates
- Unlike GROUP BY, DISTINCT doesn't necessarily produce ordered stream

SELECT DISTINCT in EXPLAIN

- One “Using temporary” even if it actually uses several temporary tables

```
explain select distinct col1 from t1;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	t1	ALL	NULL	NULL	NULL	NULL	4	Using temporary

```
explain
```

```
select distinct max(col1) from t1 group by col2;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	t1	ALL	NULL	NULL	NULL	NULL	4	Using temporary; Using filesort

Our plans for GROUP BY/ORDER BY

- WL#3220 Loose index scan for aggregate functions
- BUG#29443 make the join optimizer a *little* bit smarter about the costs (but not a full solution)

Questions

Thank you!