WL#2474

Batched range read functions

refinements to

Multi Range Read interface

Sergey Petrunia
(sergefp@mysql.com)

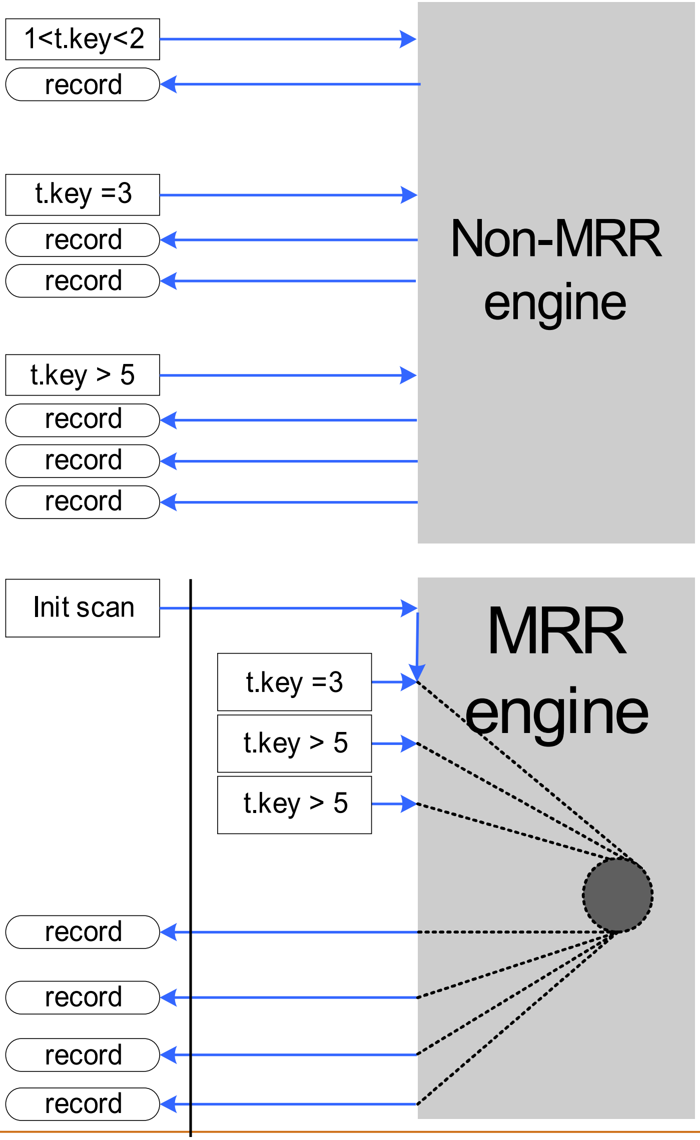
Refresh(1): The idea of MRR interface

Non-MRR

- At least one roundtrip per scanned range.
- Engine is forced to access index tuples/table records in pre-determined order.

MRR

- The number of roundtrips can be reduced to one.
- Engine can re-order table data accesses.



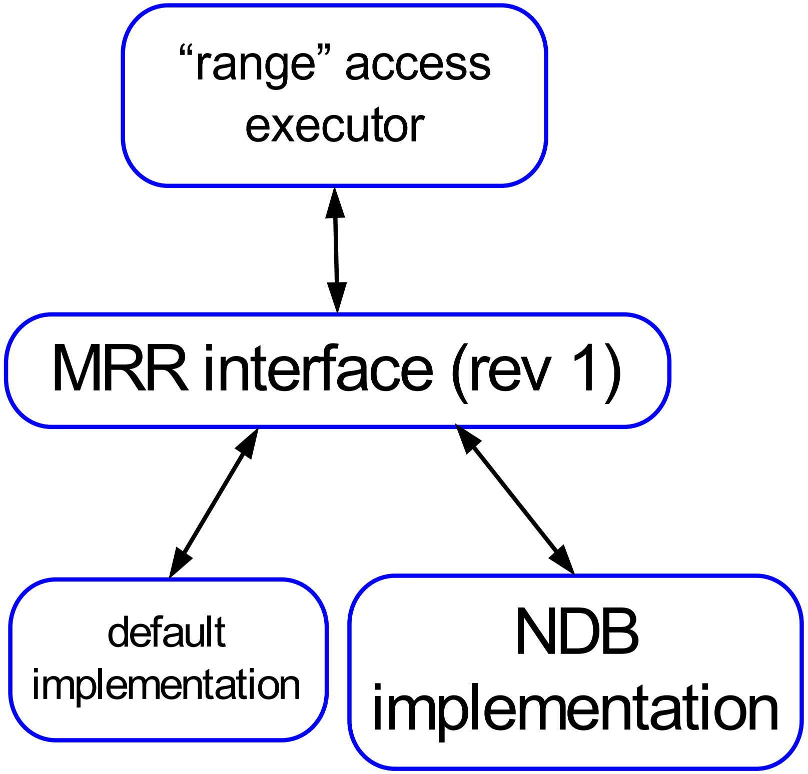
Refresh(2): MRR features in 5.0

Users

- “range” access method execution code

Implementers

- NDB
- Default (MRR-to-non-MRR converter)



MRR execution: «range» access to NDB tables

New MRR features in 5.2

New users

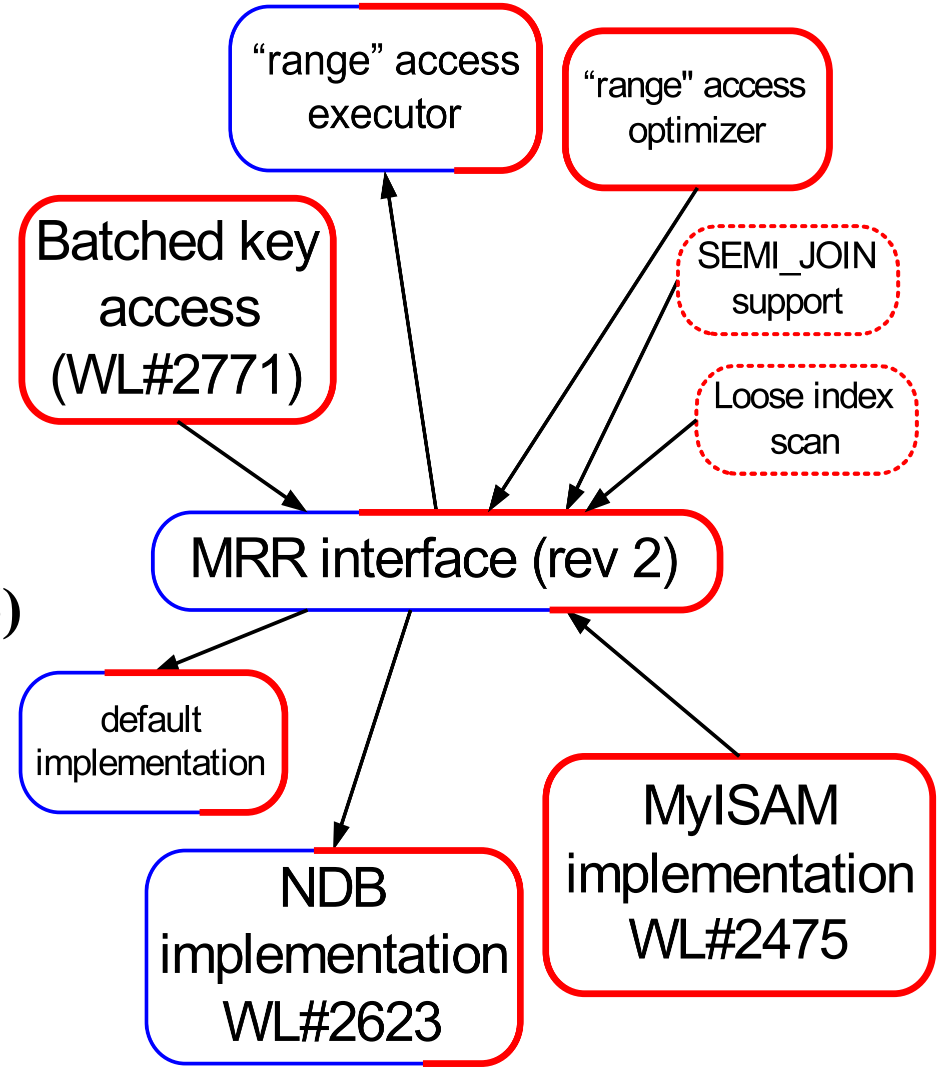
- **Batched key access (WL#2771)**
- Subquery optimization
  - Semi-join support
  - Loose index scan support
- Range access optimizer

New Implementer:

- **MyISAM & InnoDB (WL#2475)**

The above require changes in MRR interface, and "domino" changes in all affected code:

- WL#2474 (interface + default implementation)
- WL#2633 for NDB

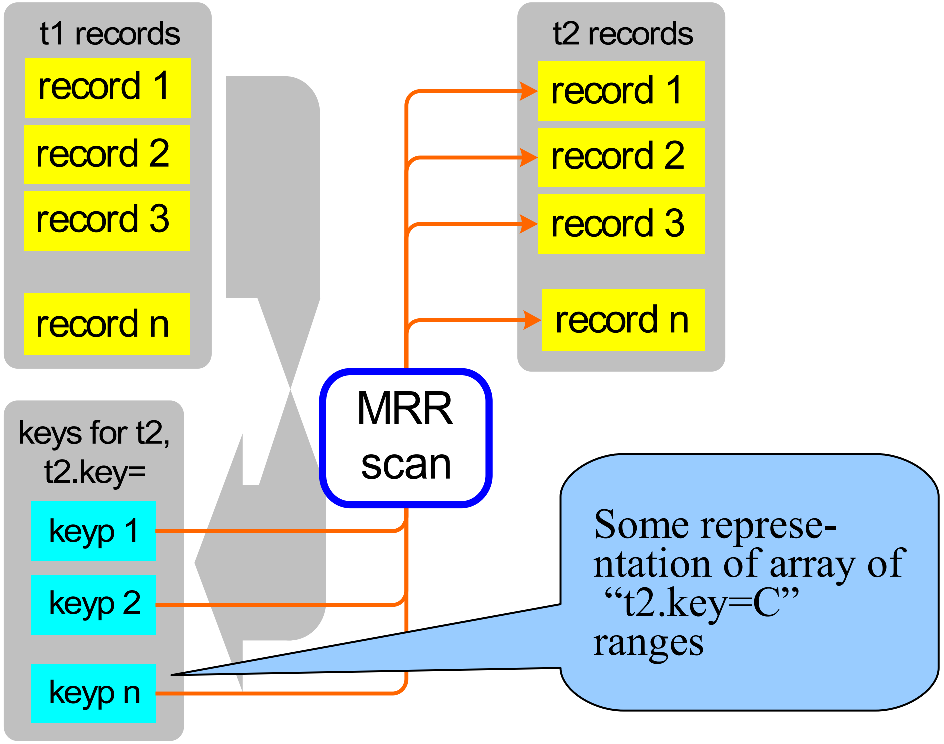


New MRR user: Batched Key Access:

```
SELECT * FROM t1,t2,...
WHERE cond1(t1) AND t2.key=t1.field AND ...
```

Non-batched	Batched

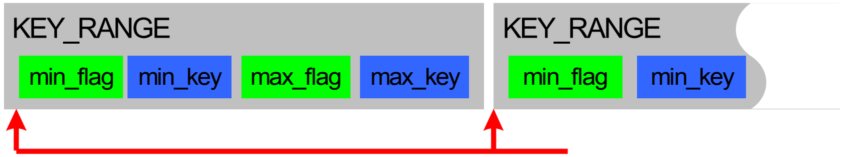
```
for each record R1 in t1 such that
      cond1(R1)
{
   t2.index_read(t2.key=R1.field);
   while (t2.index_next_same())
   {
      …
   }
}
/* At least n_matching_rows(t1)
   roundtrips in access to t2 */
```



Down to one roundtrip per join execution, depending on buffer size

New MRR user: Batched Key Access: range representations

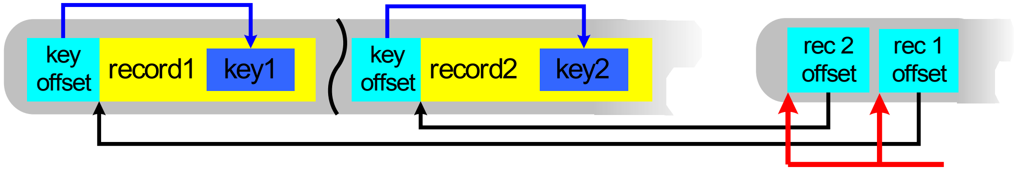
- “range”: array of “generic” interval structures



`range_cond(t.key)`

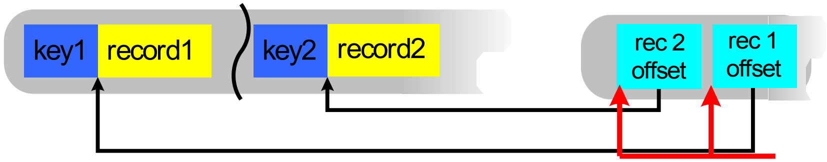
(taken from WL#2771 text)

- BKA: access key is part of previous table record



`t.key = tprev.field`

- BKA: access key is not part of previous table record



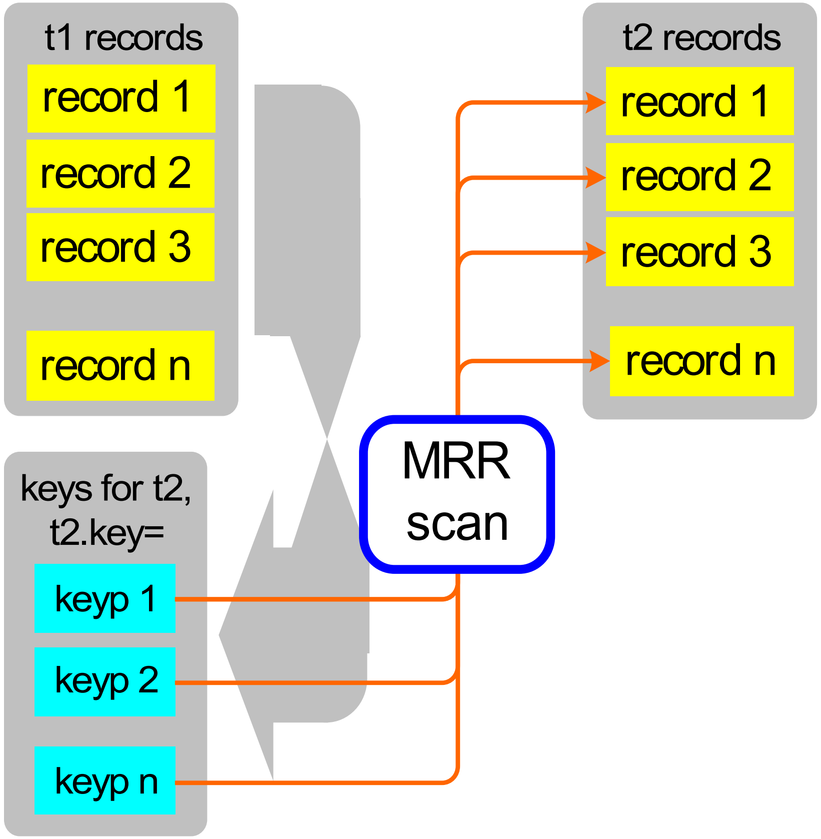
```
t.key =
func(tprev.field)
```

- BKA: there might be other layouts?

? ? ?

New MRR user: Batched Key Access: Conclusions

- Ranges to be scanned may be not known at optimization phase
- MRR user may need to know which of the ranges contains the returned record
- A frequent case: lots of intervals of the same type, and memory-efficient representation of interval is crucial.
  - ⇒ MRR implementers should not assume source intervals to be represented as in array<KEY_RANGE>



New implementer: MRR/MyISAM

Basic idea

```
read a portion of rowids into buffer;
sort the buffer by rowid;
retrieve full table records (in one "sweep");
[optional] sort the records back by key;
pass records to output;
```

Conclusions for MRR:

- Another engine with different characteristics, we need per-engine MRR scan cost functions. A cost function should be aware
  - if HA_EXTRA_KEYREAD will be used
  - if output should be sorted
  - which fields are in the output field set

New MRR user: range optimizer

Current code:

opt_range.cc,check_quick_keys():

```
produce a graph-representation;
for each interval i
{
 if (i is not a proper interval)
     return “can’t use range access”;
 nrows+= file->records_in_range(i);
 n_intervals++;
}
// the following assumes no MRR:
cost=
 file->read_time(nrows,
                              n_intervals);
```

New MRR-ized version:

```
produce a graph-representation;
iter= initialize iterator to traverse it;
(n_rows, cost)=
  file->mrr_read_info(index, iter);

ha_smth::mrr_read_info()
{
 for each interval i
 {
  if (i is not a proper interval)
      return “can’t use range access”;
    …
 }
  …
}
```

New MRR interface (1)

Range sequence interface

```
typedef struct {
  // Initialize the enumeration
  range_seq (*init)(init_params, // opaque value
                    n_ranges,    // number of ranges in
                                 // seq, obsolete
                    flags        // (see below)
                    );

  // Fill the KEY_RANGE with info about next range. (*)
  uint (*next)(range_seq, KEY_RANGE *range);
} RANGE_SEQ_IF;
```

(*) Currently all KEY_RANGE members are filled. It could be possible to analyze 'flags' parameter and avoid filling extra record.

New MRR interface (2)

Executor functions

```
int multi_read_range_init(range_seq, seq_init_param,
                          n_ranges, modes,
                          HANDLER_BUFFER *buffer);
```

range_seq, seq_init_para, n_ranges:

Range sequence iterator + number of ranges (for compatibility).

modes:

A combination of flags:

HA_MRR_SORTED

HA_MRR_INDEX_ONLY

HA_MRR_NO_ASSOCIATION

HA_MRR_{OUTER|SEMI|ANTI}_JOIN – not implemented yet

buffer:

Caller passes available buffer, callee may “return” the unused end part of the buffer

```
int multi_read_range_next(void **interval);
```

output tuple is returned in table->record[0]

New MRR interface (3)

Optimizer functions (1): unknown ranges

```
int multi_range_read_info(
    keyno,            // index to use
    n_ranges,         // E(#ranges in the sequence)
    n_rows,           // E(total #rows to read)
    uint *bufsz,      // IN:  available buffer size
                      // OUT: requested buffer size (*)
    uint *flags,      // IN: required operation modes,
                      // OUT: actually used modes (**)
    COST_VECT *cost   // OUT: total scan cost
);
```

(*) the WL# HLS says a handler may request a buffer bigger then was provided but no MRR user support it.

(**) MRR implementation may 'refuse to sort' by clearing HA_MRR_SORTED

New MRR interface (4)

Optimizer functions (2): known ranges

The same function as previous but it accepts a concrete range interval

```
ha_rows multi_range_read_info_const(
  keyno,           // index to use
  ranges_seq,      // sequence of source range
  seq_init_param,
  uint *bufsz,     // IN:  available buffer size
                   // OUT: requested buffer size
  uint *flags,     // IN: required operation modes,
                   // OUT: actually used modes (**)
  COST_VECT *cost // OUT: total scan cost
);
```

Note:

- The interval enumeration may “fail” in the middle.

New MRR interface (4)

- **Interoperability with condition pushdown**
  Assume cond_push() has been called before any MRR calls, including optimizer calls.
- **Requested fields set**
  All MRR functions shall assume that “read fieldset” is set up appropriately at the time they are invoked.

Unclear issues

- **Cost of sorting**
  Suppose MRR implementation can produce sorted output, but at some additional cost. How can we decide whether we should let it sort or do sorting in SQL layer with filesort()?
  (if MRR implementation can sort it should do it as it can use e.g. “merge the streams” sort which it can do cheaper then filesort()?).
- **Needed memory per record**
  Given unlimited buffer size, we’ll need X bytes per record…

New MRR interface: summary

Range sequence

```
RANGE_SEQ_IF {
    (*init)();
    (*next)();
}
```

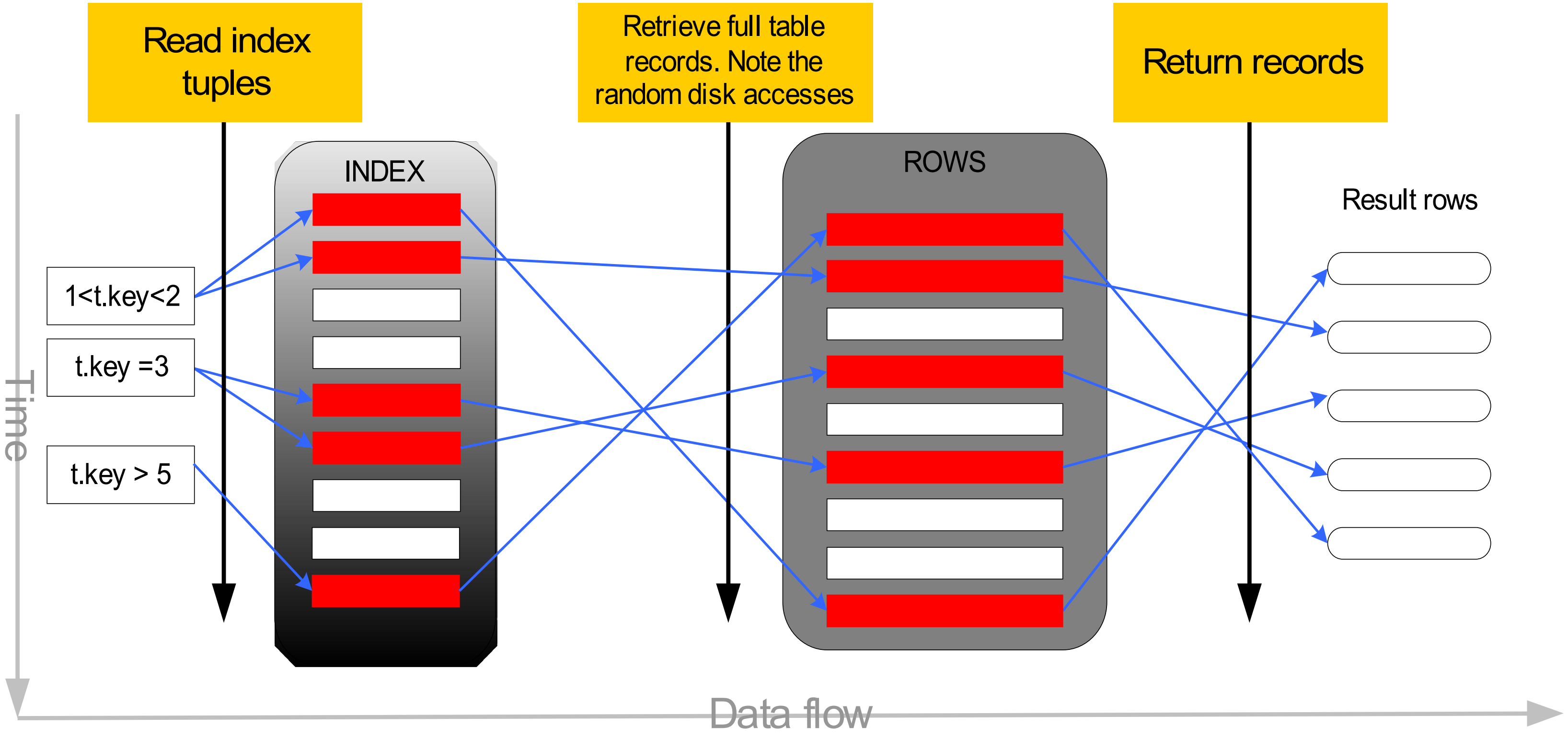
Optimizer

```
multi_range_read_info();
multi_range_read_info_const();
```

Executor

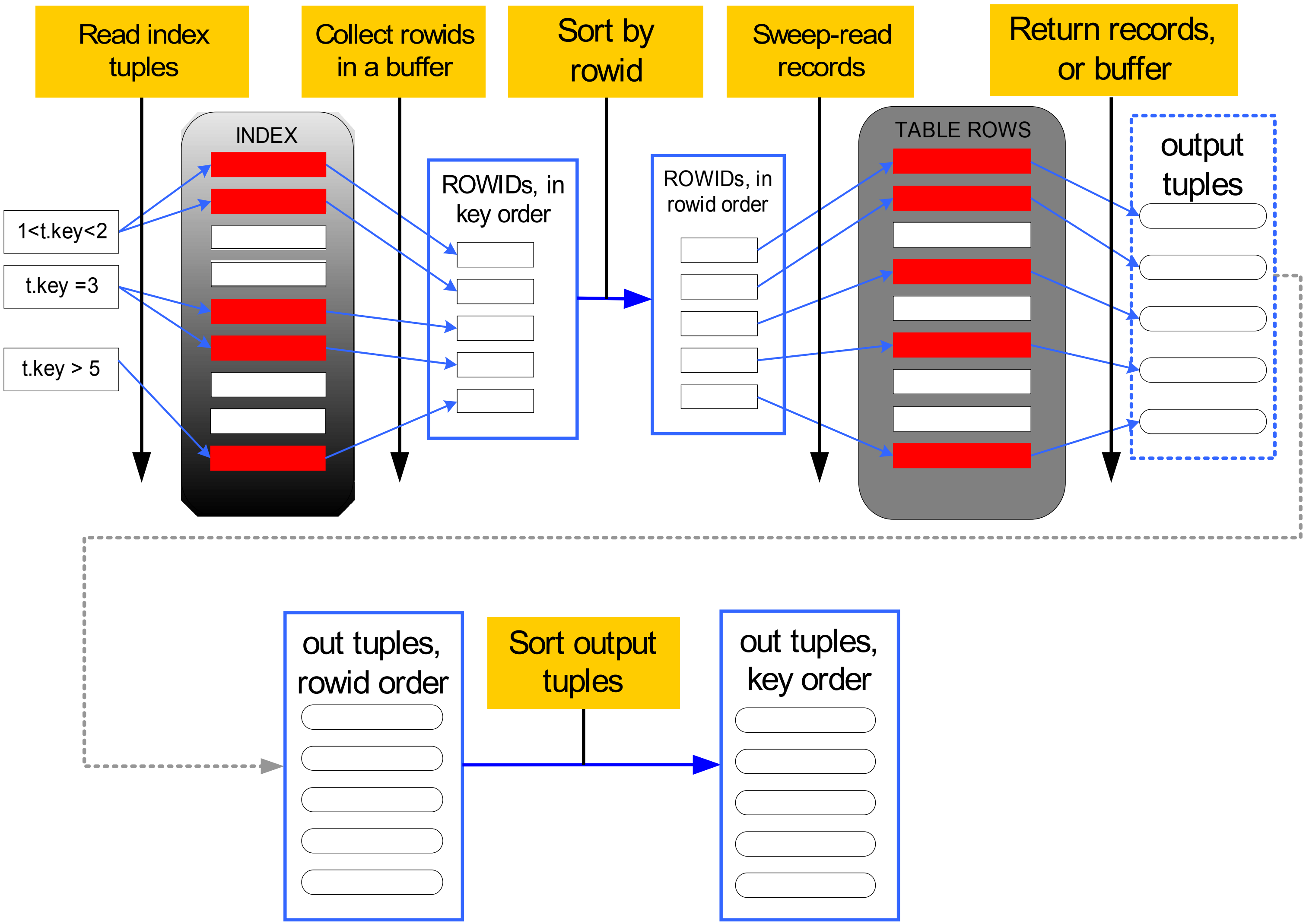
```
int multi_read_range_init();
int multi_read_range_next();
```

MyISAM operation without MRR



All disk seeks are random seeks

MRR/MyISAM operation



MRR/MyISAM : Unordered output

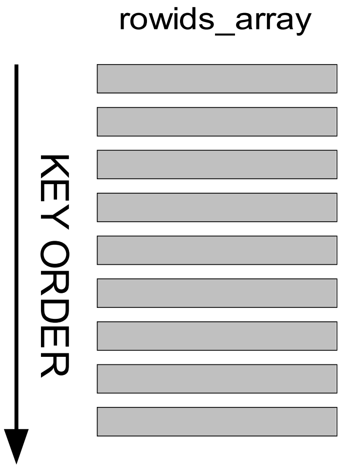
```
multi_range_read_init()
{
  fill_rowids_buffer_and_sort_it();
  /* adjust the buffer if we’ve scanned through all intervals */
  return 0;
}

multi_range_read_next()
{
  if (!have-rowids-in-the-buffer &&
    fill_rowids_buffer_and_sort_it())
    return EOF;
  return read_full_row(buffer.pop_front());
}
```

MRR/MyISAM: ordered output

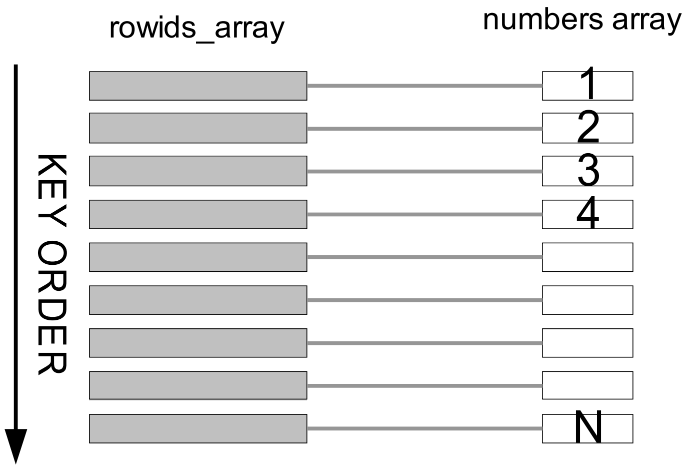
- Need to
  - buffer output tuples (buffer filled in rowid order)
  - return output tuples in key order

=> For dynamic-size output tuples need to predict output tuple sizes, so we know how many rowids to process in a "batch".

- Step1: read rowids

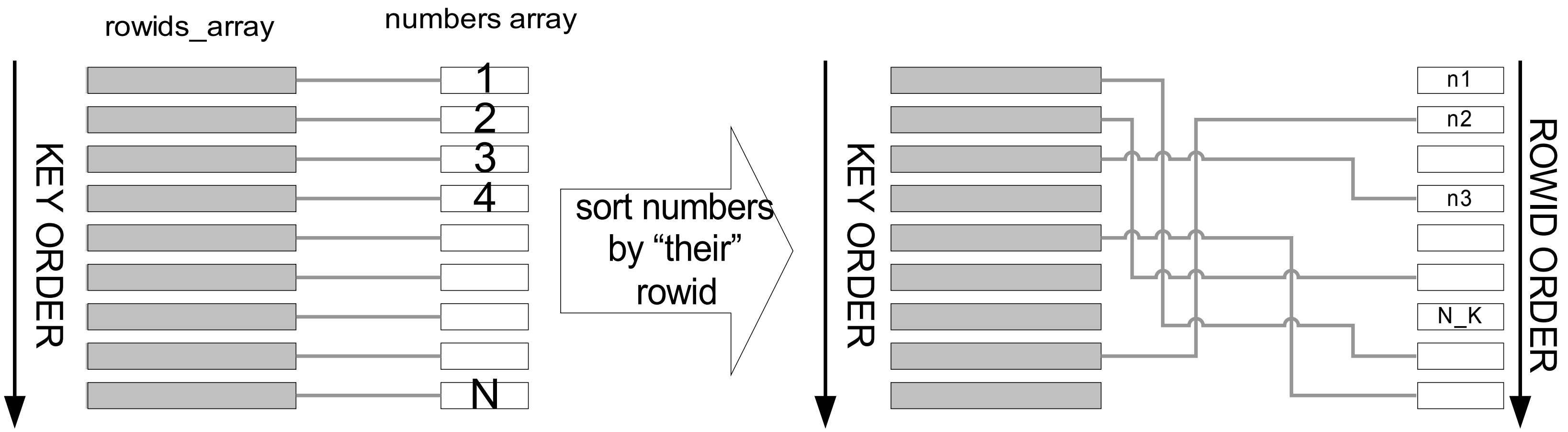


- Step2: create 1…N numbers array



MRR/MyISAM – ordered output (2)

- Step3: sort numbers_array by rowid

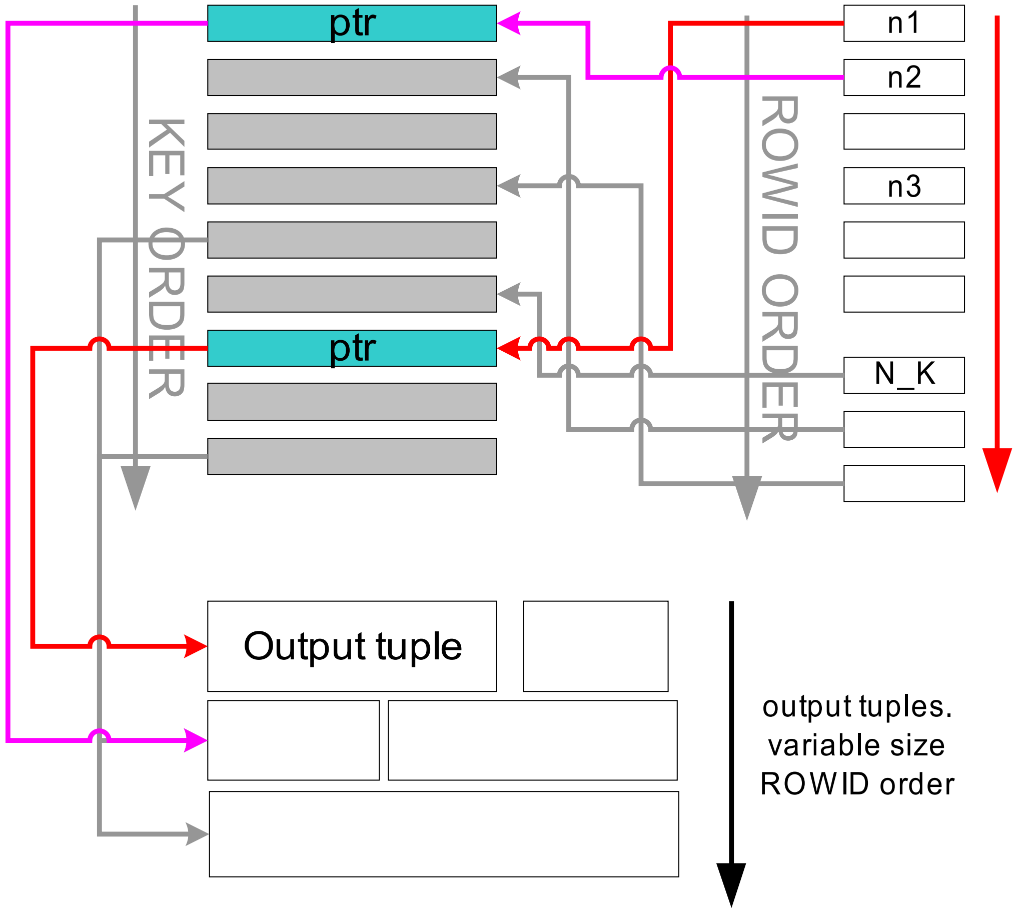


MRR/MyISAM – ordered output (3)

Step4:

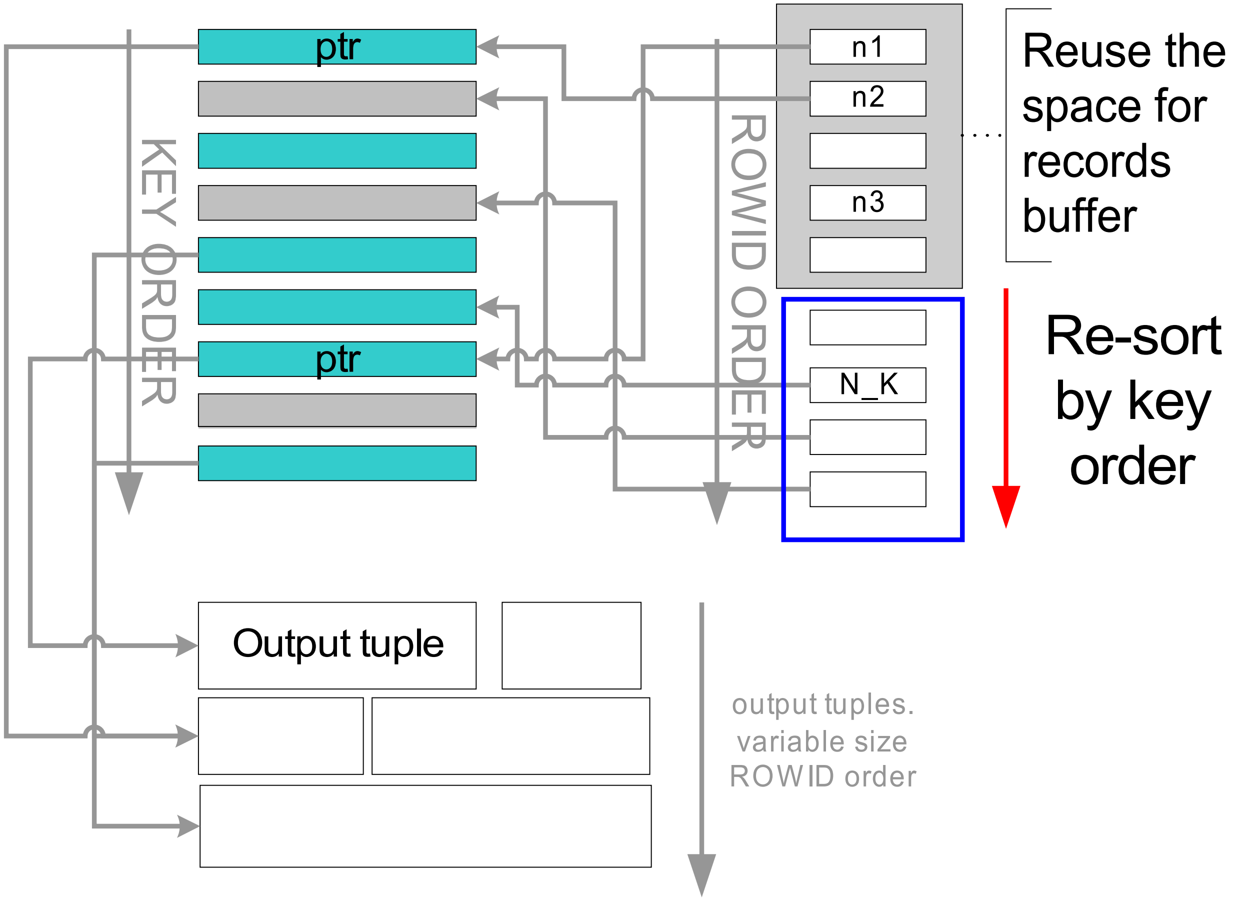
In rowid order, while space permits:

- read row into rows buffer
- replace rowid with buffer pointer



MRR/MyISAM – ordered output (4)

- Step5: re-sort the “remainder” numbers by key order

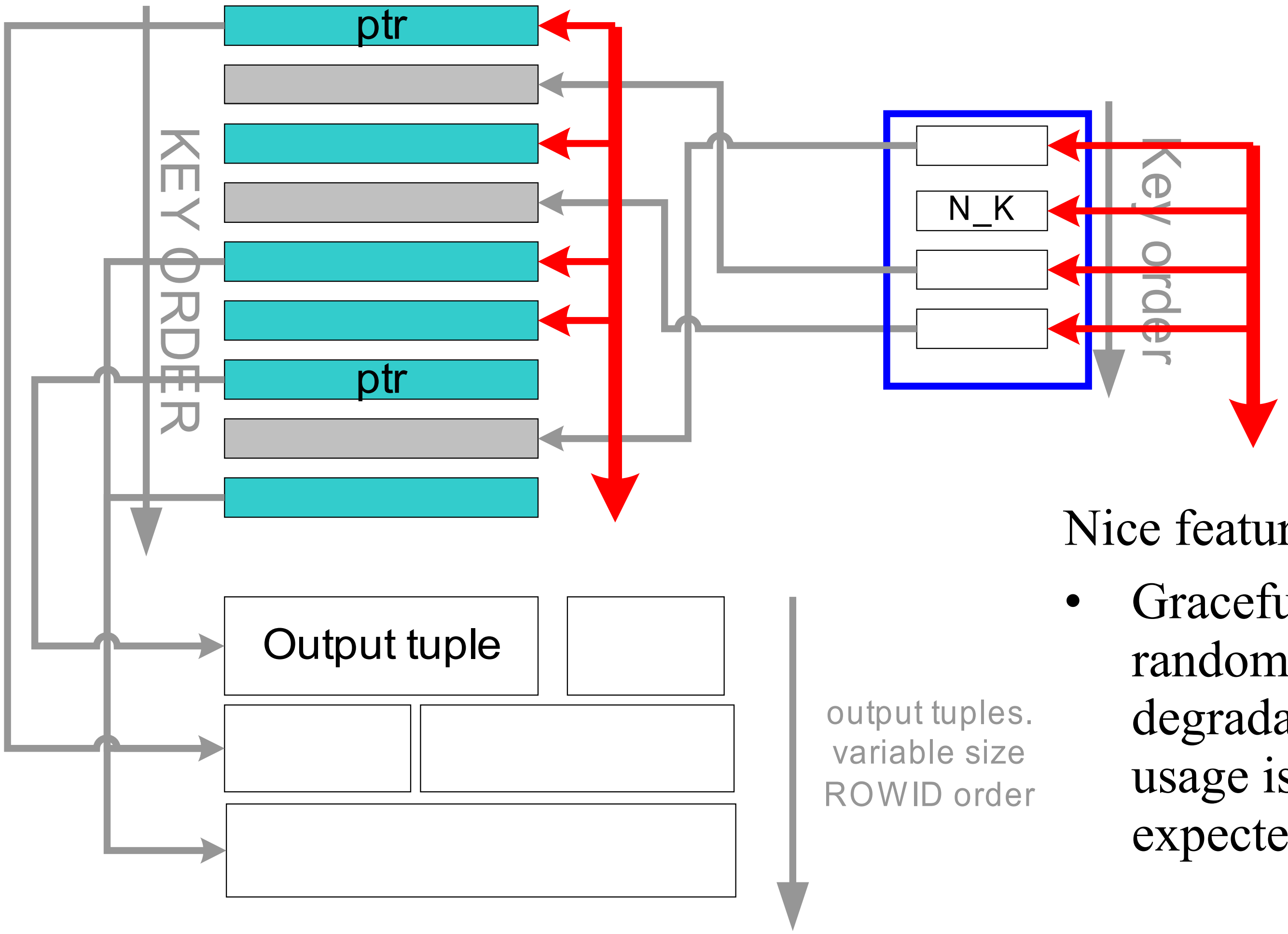


Reuse the space for records buffer

Re-sort by key order

MRR/MyISAM – ordered output (5)

- Step6: Merge two key-ordered sequences



Nice feature:

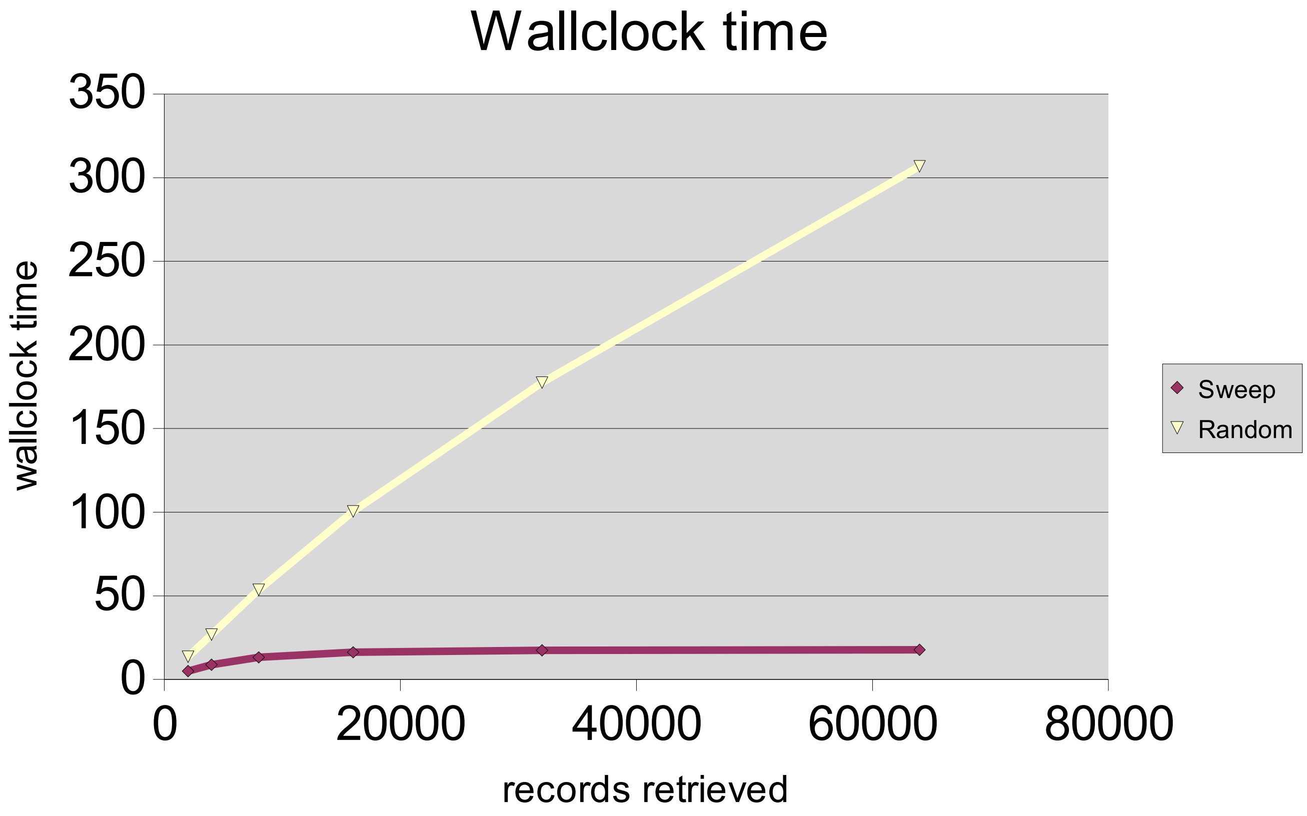
- Graceful sweep -> random-walk degradation if memory usage is higher then expected

MRR/MyISAM: preliminary benchmark

- Simulation of MRR with index_merge over specifically crafted MyISAM table.

Benchmark Paramete

- 1GB datafile
- 1 GB RAM
- MySQL key cache holds the indexes
- OS Cache flushed before each query



Observations

- Results depend **a lot** on whether the data is in cache
- “Sweep-interrupting” disk activity reduces the difference somewhat.