

MySQL Conference & Expo 2011



Why Are The New Optimizer Features Important and How Can I Benefit From Them?

Igor Babaev

igor@askmonty.org

Sergey Petrunya

psergey@askmonty.org

New query optimizer features



- We'll talk about
 - Fresh finished query optimizer features in MariaDB
- General development theme
 - Big data (IO-bound loads)
 - Big queries
 - Joins
 - SELECTs that retrieve lots of rows

New query optimizer features



- Correct optimization of `index_merge` vs `range`
- `index_merge/sort-intersect`
- Batched Key Access [improvements]
- Hash join

Benchmarking setup



- DBT-3 scale=10
 - XtraDB storage engine
 - 28 GB total dataset
 - Biggest table: 'lineitem' 60M rows, 22G on disk
- 'ontime': table from transtats.bts.gov dataset
 - XtraDB storage engine
 - Subset: flights in January-April 2009
 - 1.5 M rows, 832MB.
- innodb_buffer_pool=256M unless said otherwise

New query optimizer features



- **Correct optimization of index_merge vs range**
- index_merge/sort-intersect
- Batched Key Access [improvements]
- Hash join

index_merge vs range optimization



- A long-known, much-complained problem in MySQL:

```
MySQL [ontime]> explain select * from ontime where (Origin='SEA' or Dest='SEA');
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	ontime	index_merge	Origin, Dest	Origin, Dest	6, 6	NULL	92850	Using union(Origin, Dest); Using where

```
MySQL [ontime]> explain select * from ontime where (Origin='SEA' or Dest='SEA') and securitydelay=0;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	ontime	ref	Origin, Dest, SecurityDelay	SecurityDelay	5	const	791546	Using where

10x more. Ouch!

```
MySQL [ontime]> explain select * from ontime where (Origin='SEA' or Dest='SEA') and depdelay < 12*60;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	ontime	ALL	Origin, DepDelay, Dest	NULL	NULL	NULL	1583093	Using where

15x more. Ouch!

- => Additional AND-ed predicates cause index_merge plan to be removed from consideration, and a worse plan to be chosen
 - Slowdown can be 10x, 100x, ...

index_merge vs range optimization



- Now, lets try the same in MariaDB 5.3:

```
MariaDB [ontime]> explain select * from ontime where (Origin='SEA' or Dest='SEA');
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	ontime	index_merge	Origin, Dest	Origin, Dest	6, 6	NULL	92800	Using union(Origin, Dest); Using where

```
MariaDB [ontime]> explain select * from ontime where (Origin='SEA' or Dest='SEA') and securitydelay=0;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	ontime	index_merge	Origin, Dest, SecurityDelay	Origin, Dest	6, 6	NULL	92800	Using union(Origin, Dest); Using where

Will still use index_merge

```
MariaDB [ontime]> explain select * from ontime where (Origin='SEA' or Dest='SEA') and depdelay < 12*60;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	ontime	index_merge	Origin, DepDelay, Dest	Origin, Dest	6, 6	NULL	92800	Using union(Origin, Dest); Using where

Same here

index_merge vs range optimization



We call it

“Fair choice between range and index_merge optimizations”

- It is in MariaDB 5.3
- New functionality is always on (no way to switch to old behavior)
- No known problems.

New query optimizer features



- Correct optimization of `index_merge` vs `range`
- `index_merge/sort-intersect`
- Batched Key Access
- Hash join

index_merge/sort_intersect



MySQL and MariaDB 5.{1,2} support index_merge/intersection:

```
MySQL [ontime]> explain select avg(arrdelay) from ontime where depdel15=1 and OriginState = 'CA';
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|id|select_type|table |type      |possible_keys      |key          |key_len|ref  |rows |Extra
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1|SIMPLE      |ontime|index_merge|OriginState,DepDel15|OriginState,DepDel15|3,5    |NULL|76952|Using intersect (OriginState,DepDel15);Using where
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
# The select takes 2.20 sec
```

But it only works with equality conditions:

```
MySQL [ontime]> explain select avg(arrdelay) from ontime where depdel15=1 and OriginState IN ('CA', 'GB');
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|id|select_type|table |type|possible_keys      |key          |key_len|ref  |rows |Extra
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1|SIMPLE      |ontime|ref  |OriginState,DepDel15|DepDel15|5      |const|36926|Using where
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
# The select takes 10.78 sec
```

There are no flights from 'GB' state. Still, the query is 5x slower.

MariaDB 5.3 doesn't have this limitation:

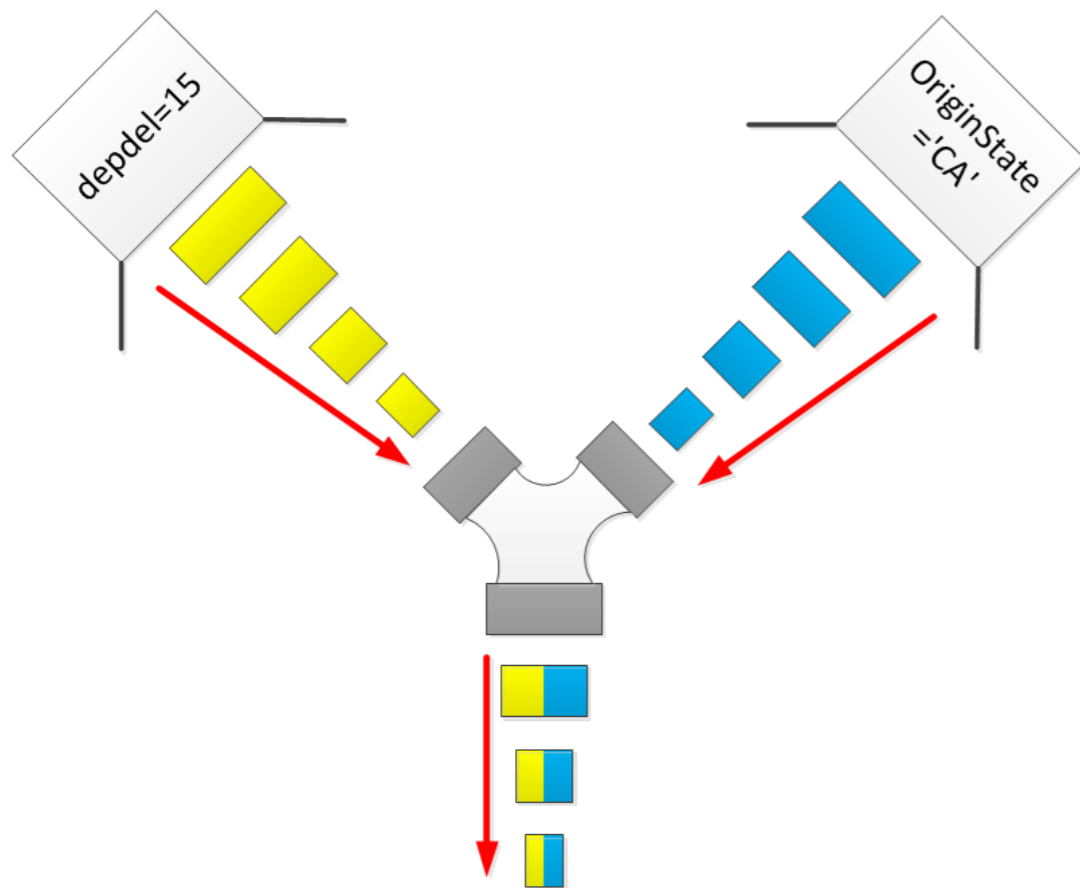
```
MariaDB [ontime]> explain select avg(arrdelay) from ontime where depdel15=1 and OriginState IN ('CA', 'GB');
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|id|select_type|table |type      |possible_keys      |key          |key_len|ref  |rows |Extra
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1|SIMPLE      |ontime|index_merge|OriginState,DepDel15|DepDel15,OriginState|5,3    |NULL|60754|Using sort_intersect (DepDel15,OriginState); Using where
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
# The select takes 3.23 sec
```

Intersect vs sort_intersect

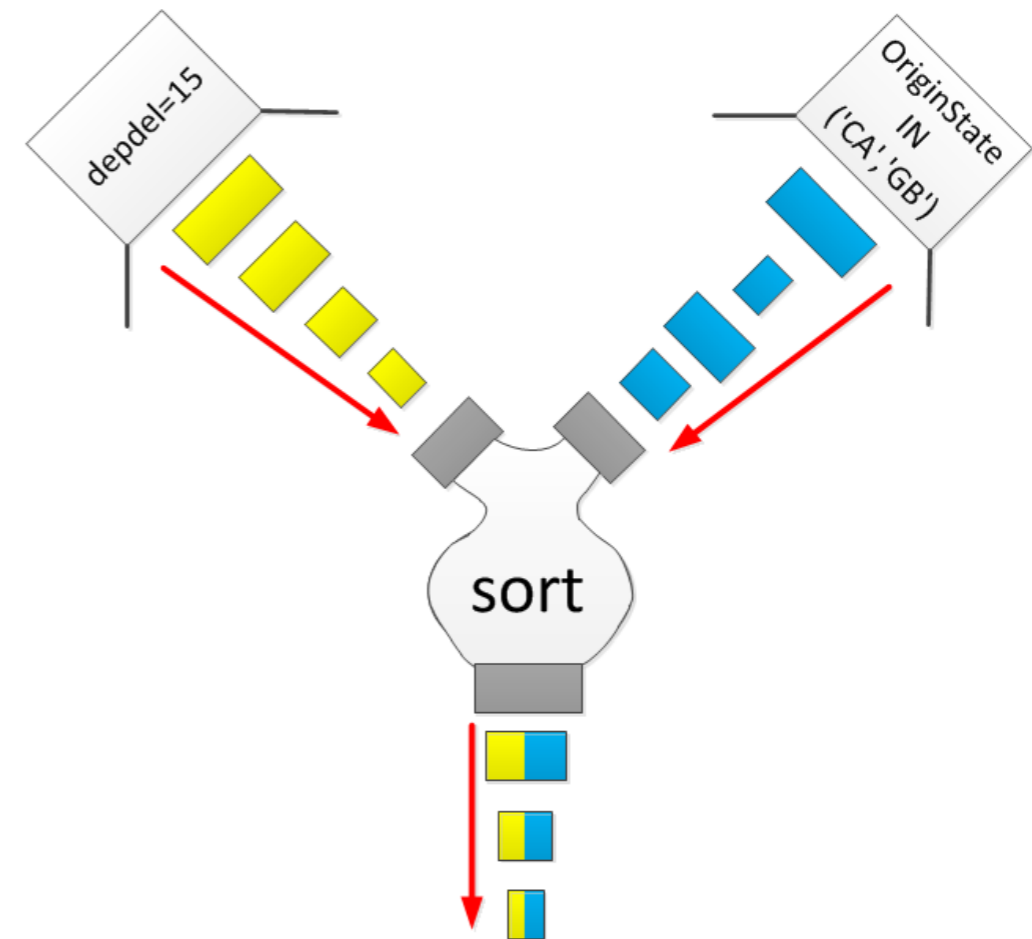


- Different execution strategies

Regular intersect merges ordered streams



Sort-intersect doesn't require inputs to be ordered



Benchmarking sort-intersect



```
set optimizer_switch='index_merge_sort_intersection=on';
```

```
explain
```

```
select max(l_extendedprice)
```

```
from lineitem
```

```
where
```

```
l_partkey between 1 and 50000 and
```

```
l_shipdate between '1993-07-01' and '1993-07-31';
```

```
***** 1. row *****
```

```
id: 1
```

```
select_type: SIMPLE
```

```
table: lineitem
```

```
type: index_merge
```

```
possible_keys: i_l_shipdate,i_l_suppkey_partkey,i_l_partkey
```

```
key: i_l_shipdate,i_l_partkey
```

```
key_len: 4,5
```

```
ref: NULL
```

```
rows: 35823
```

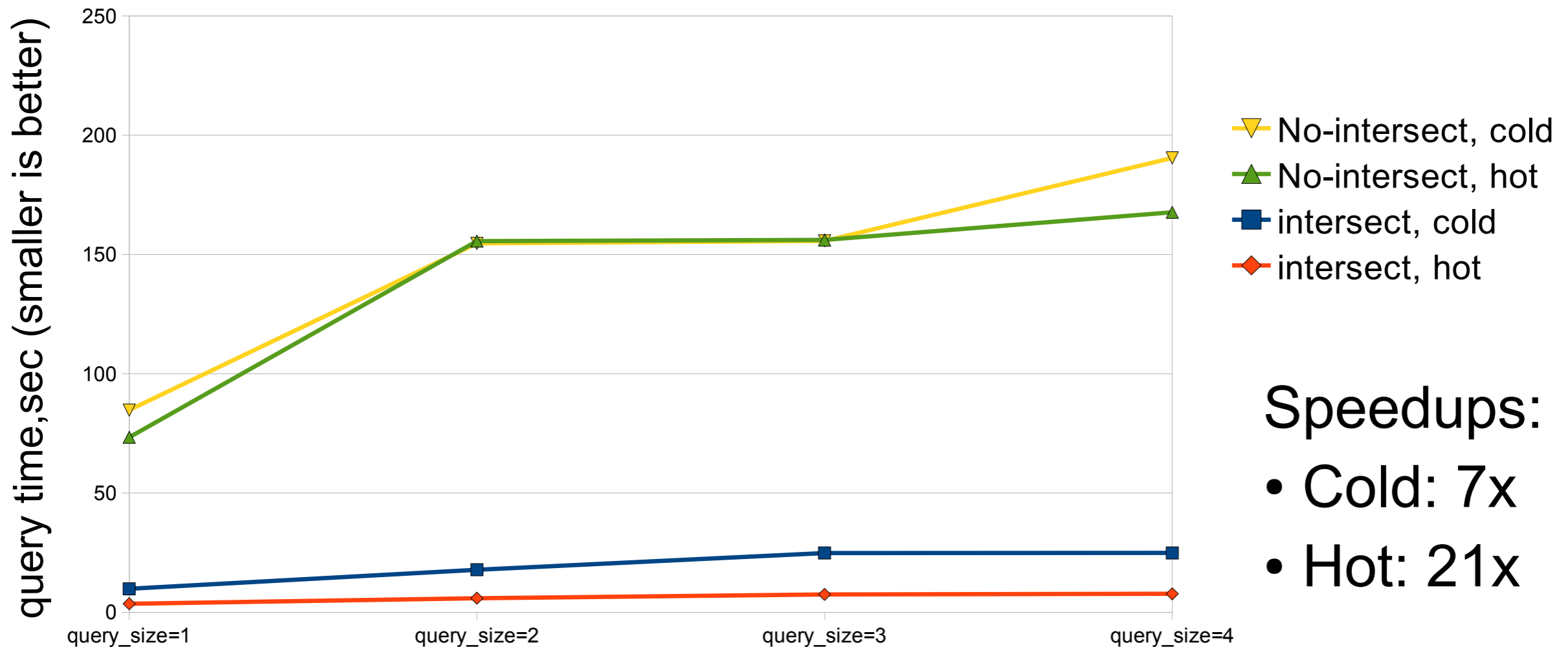
```
Extra: Using sort_intersect(i_l_shipdate,i_l_partkey); Using where  
1 row in set (0.01 sec)
```

Benchmarking sort-intersect



Test runs

- hot/cold buffer pool,
- 4 queries with varying range sizes



Speedups:

- Cold: 7x
- Hot: 21x

New query optimizer features



- Correct optimization of index_merge vs range
- index_merge/sort-intersect
- **Batched Key Access**
- Hash join

Batched Key Access



- Batched Key Access feature (“BKA”)
 - Originally in (never released) MySQL 6.0
 - Now in MariaDB 5.3
 - With EXPLAIN showing more information
 - A number of bugs (the “MRR bug pile”) were fixed
 - New improvement “Key ordered retrieval”



Nested Loop Join

Nested loops join



table1

A vertical rectangle representing a table with many rows. The top and bottom rows are white. There are six red horizontal bars, one in each of the following row groups: the 10th row, the 20th row, the 30th row, the 40th row, the 50th row, and the 60th row. All other rows are white.

table2

A vertical rectangle representing a table with many rows, all of which are white.

Nested loops join



table1

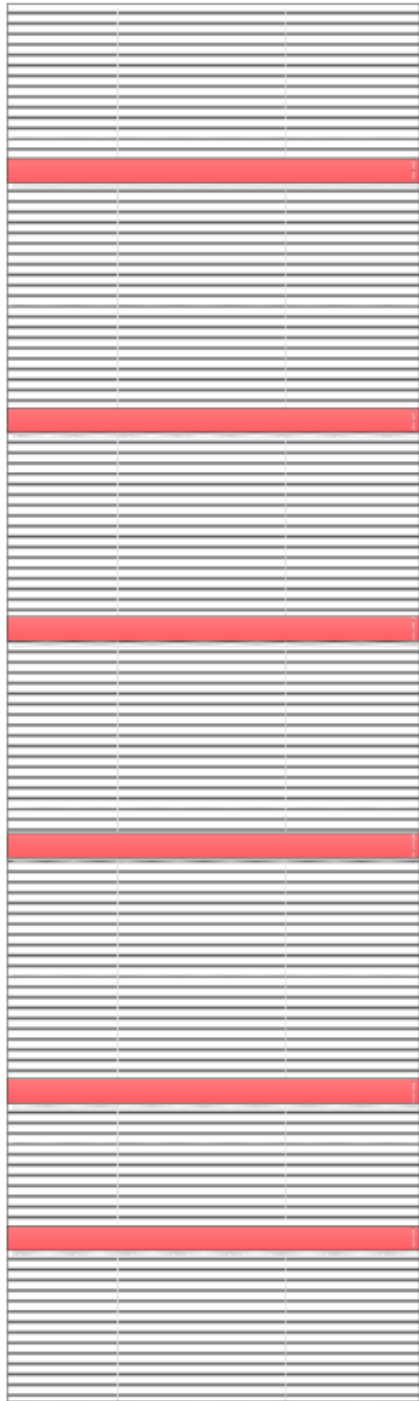
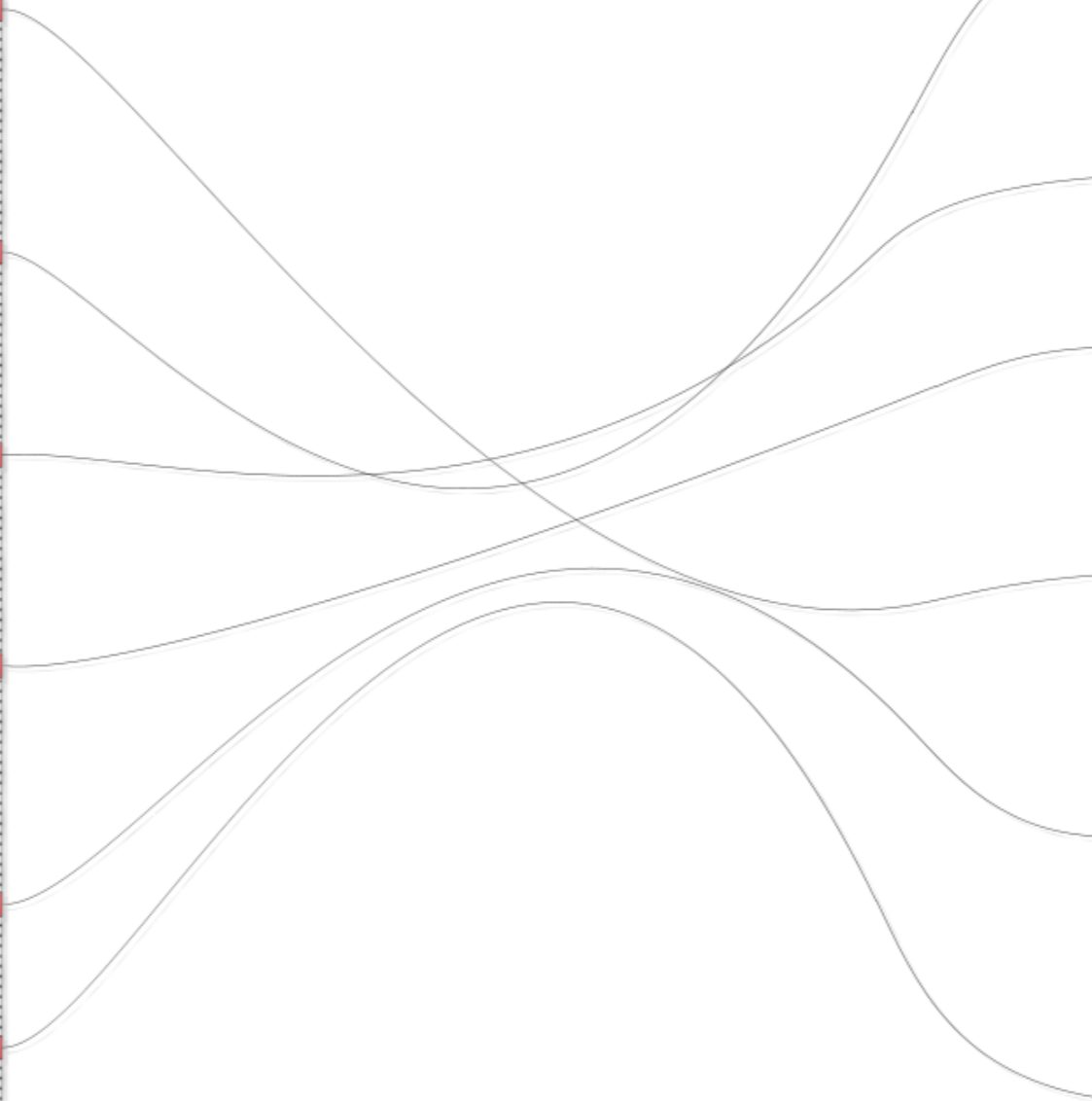
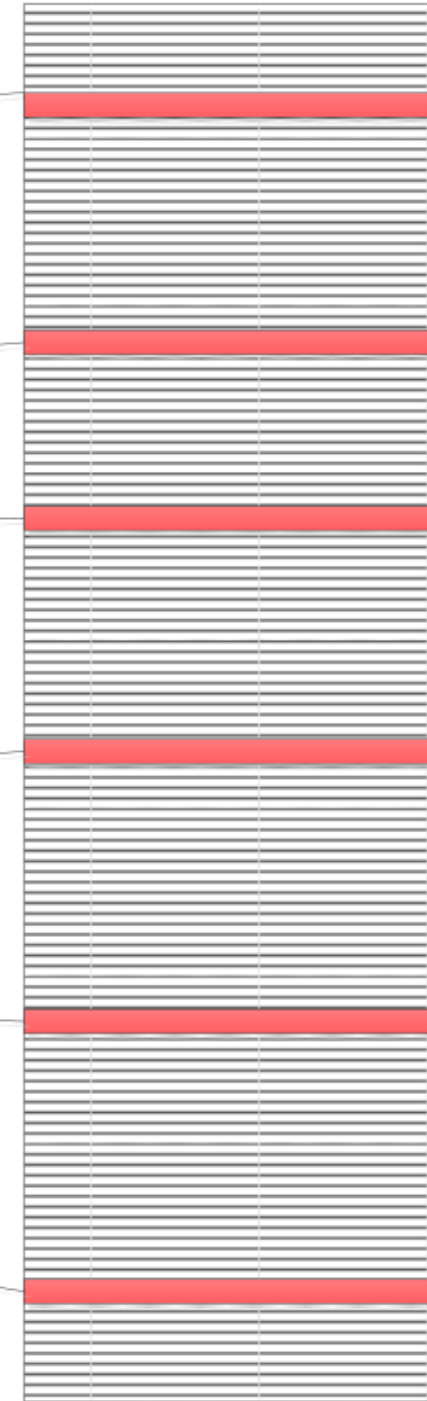
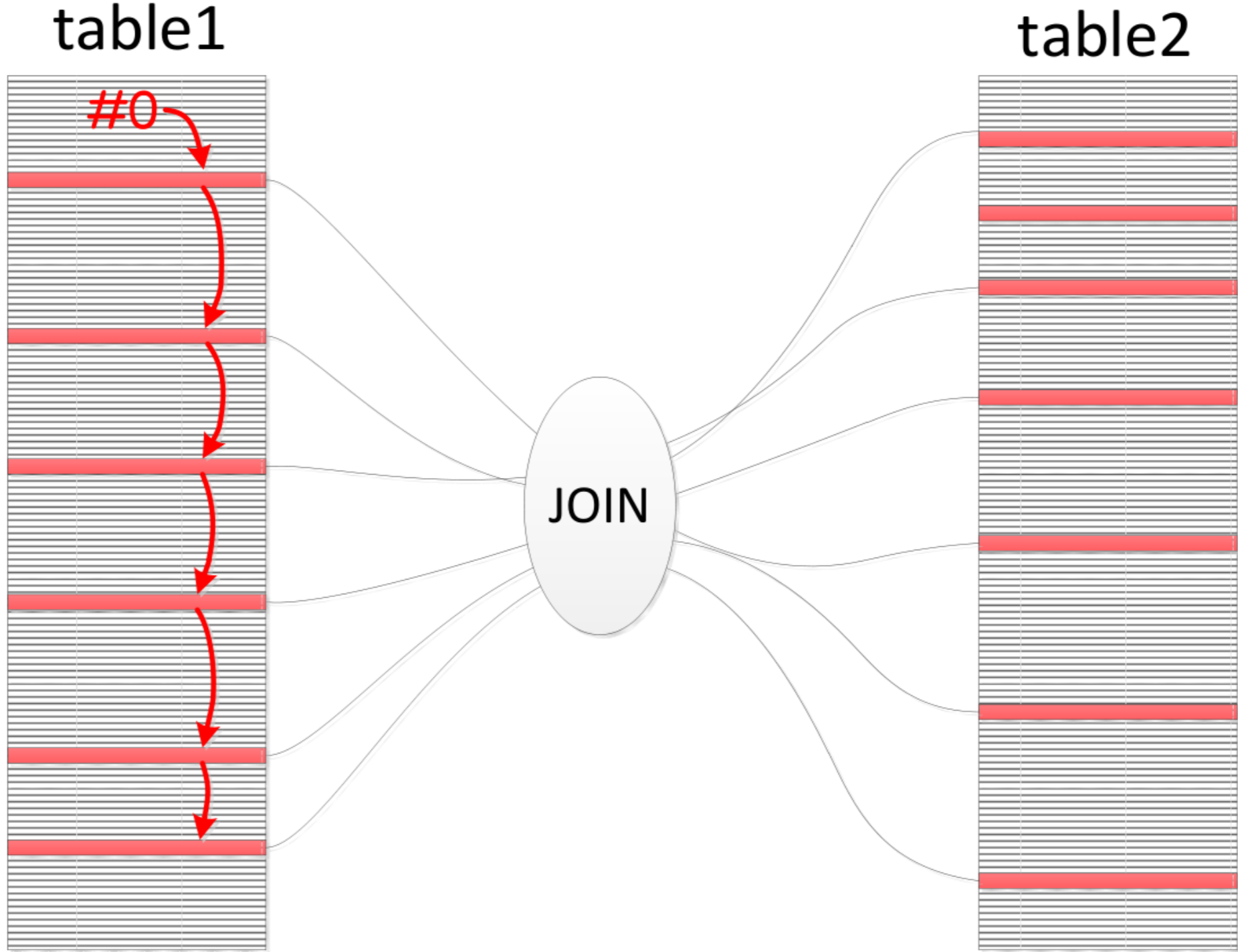


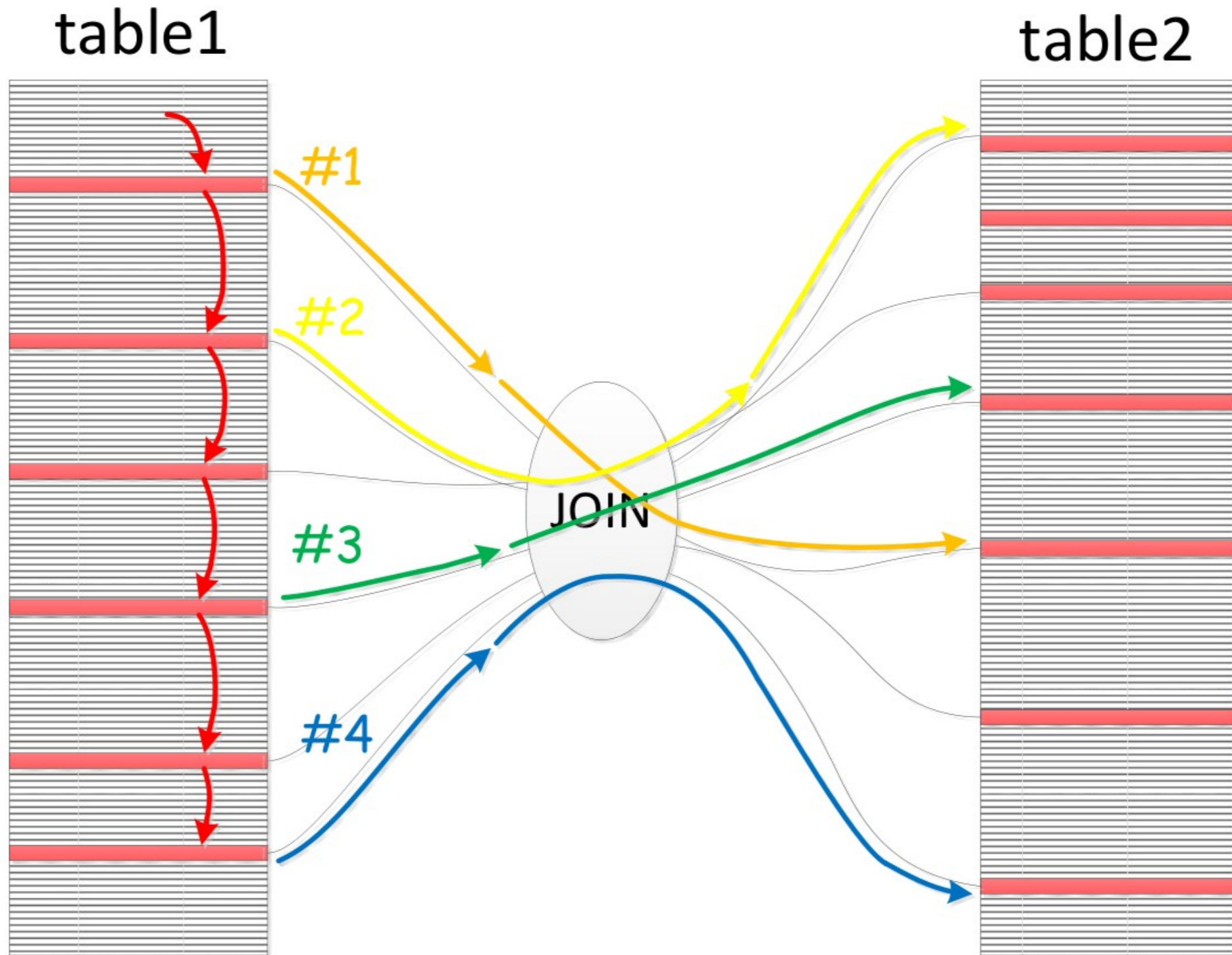
table2



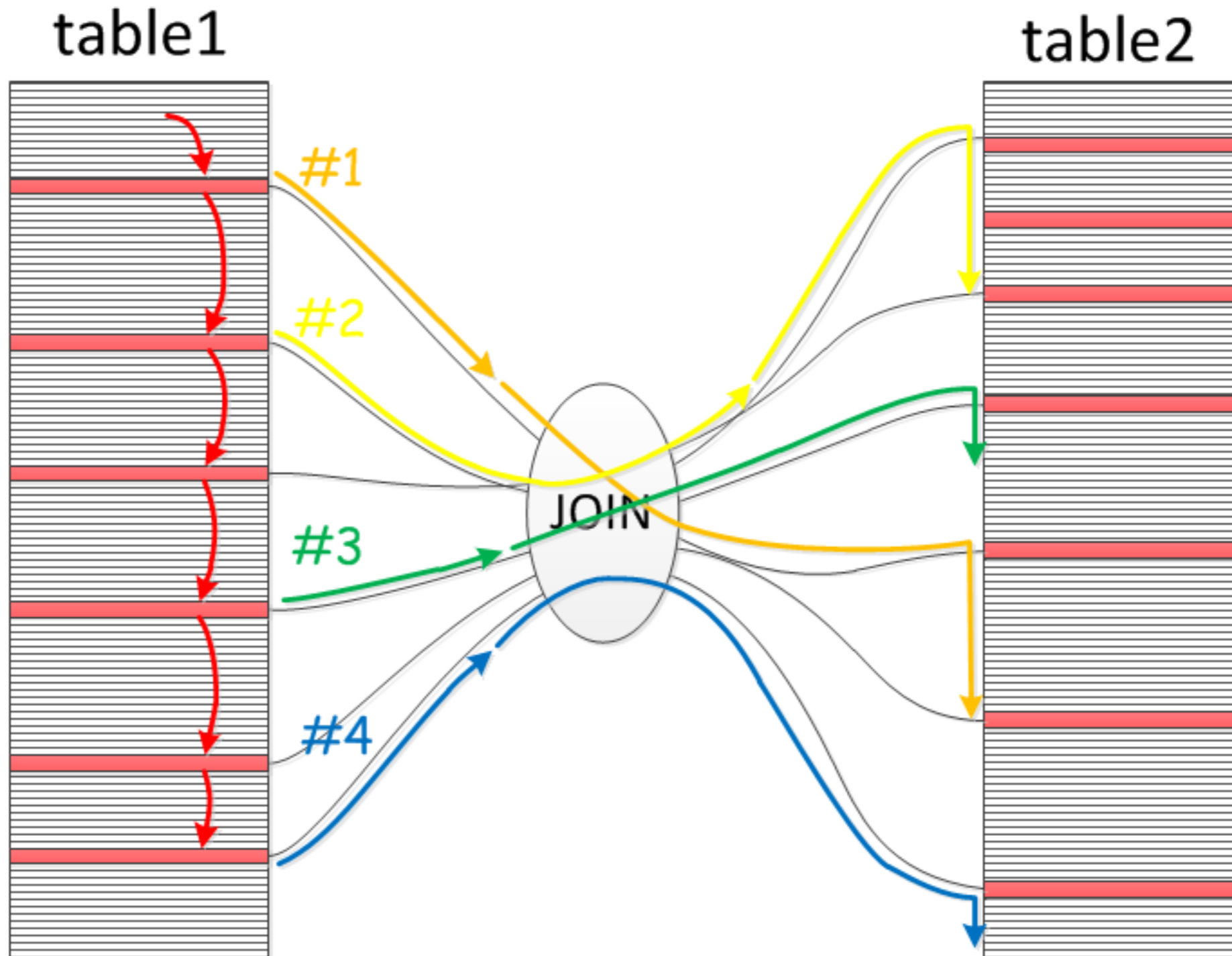
Nested loops join: outer loop



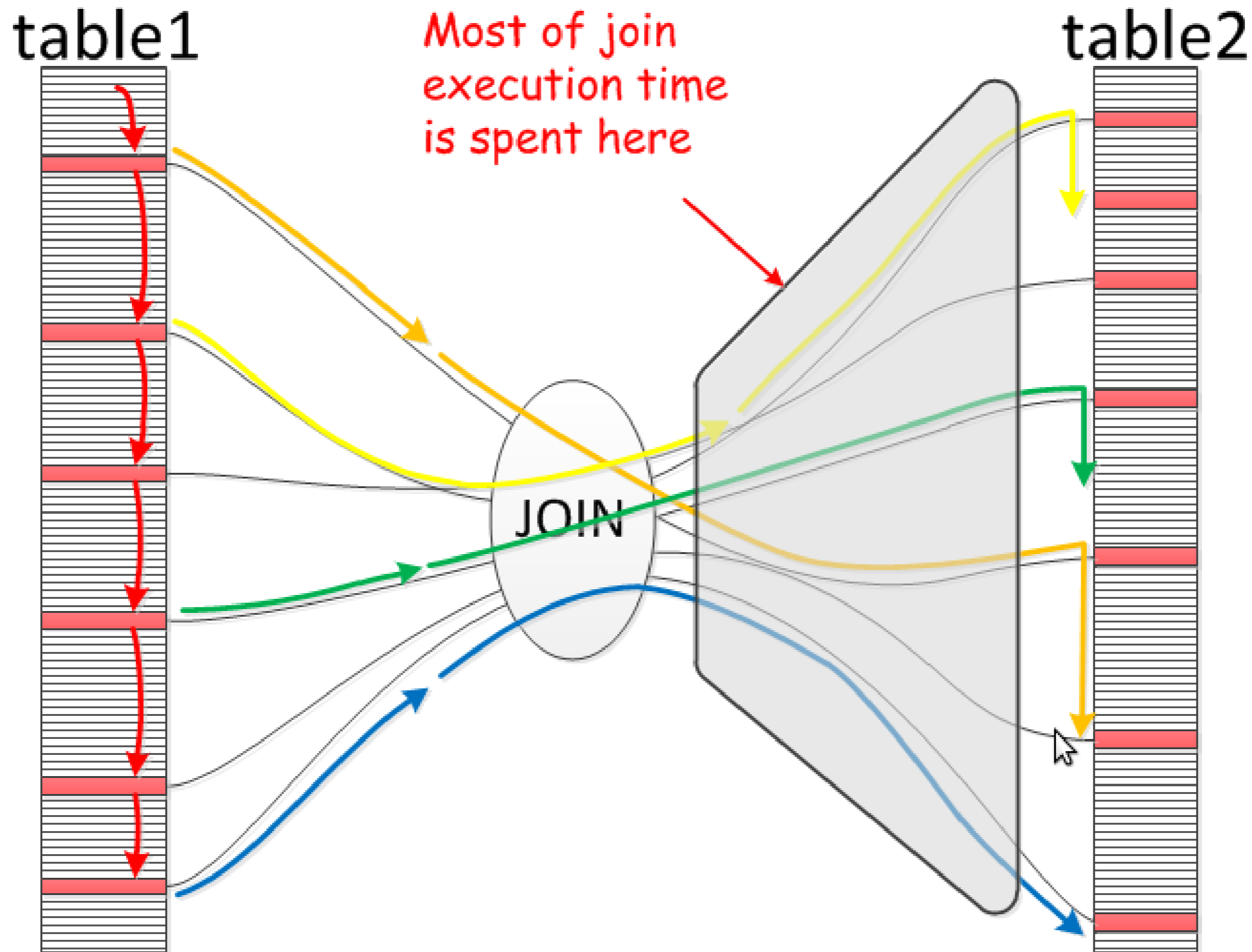
Nested loops join: lookups



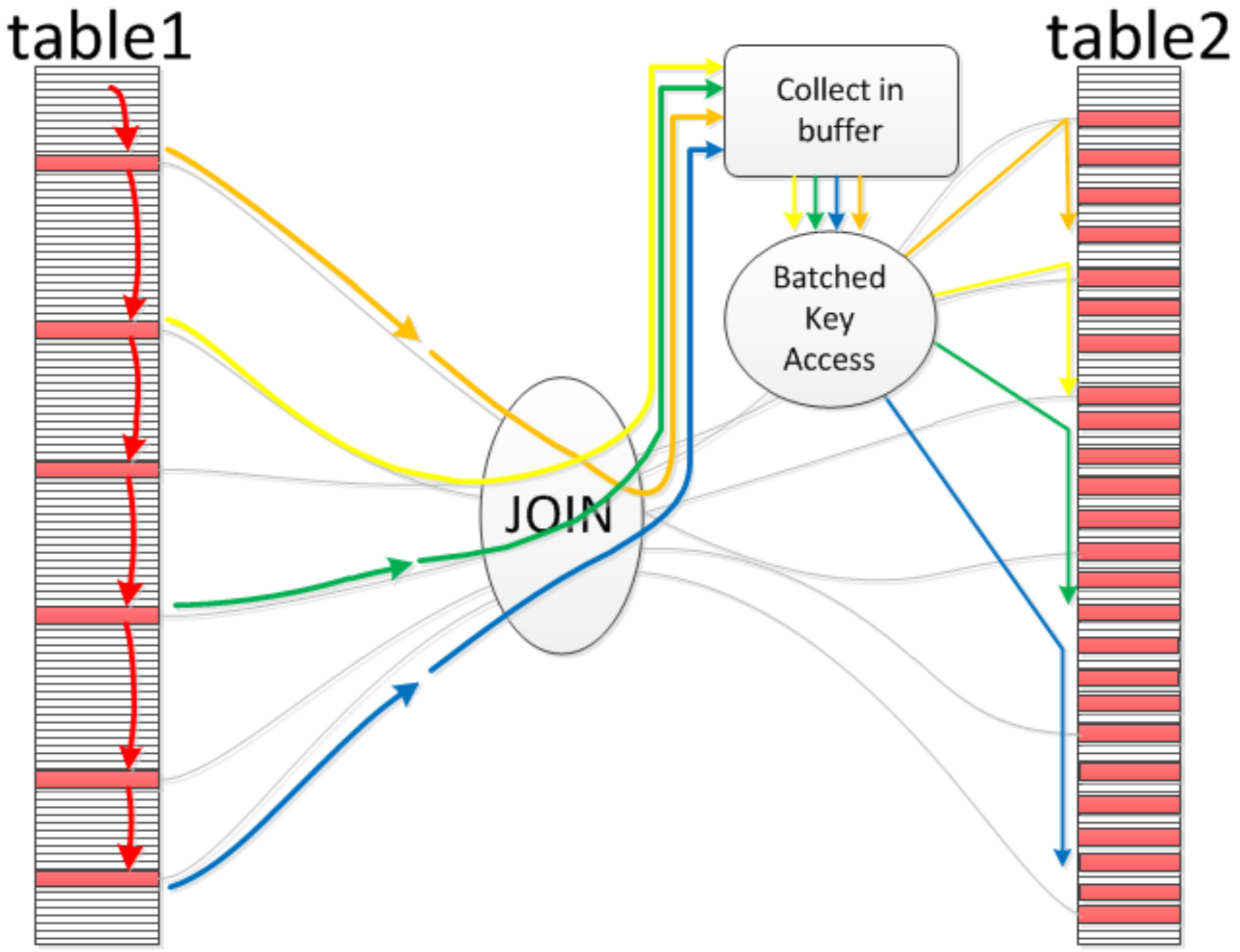
Nested loops join: inner loops



BKA and Hash Join: background



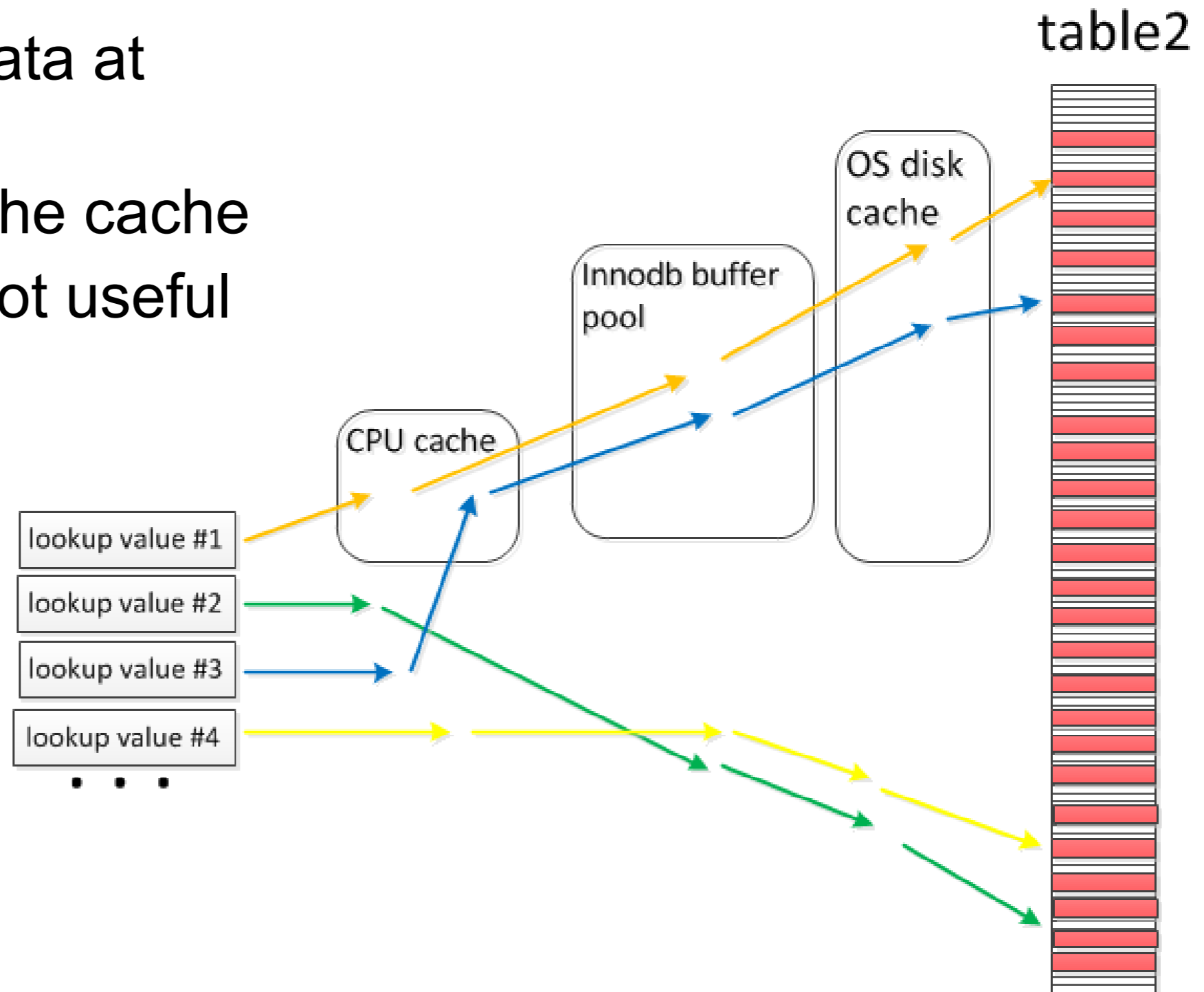
Batched Key Access idea



Why are batched index lookups faster?



- Non-BKA hits data at random
- That's hard on the cache
- Prefetching is not useful

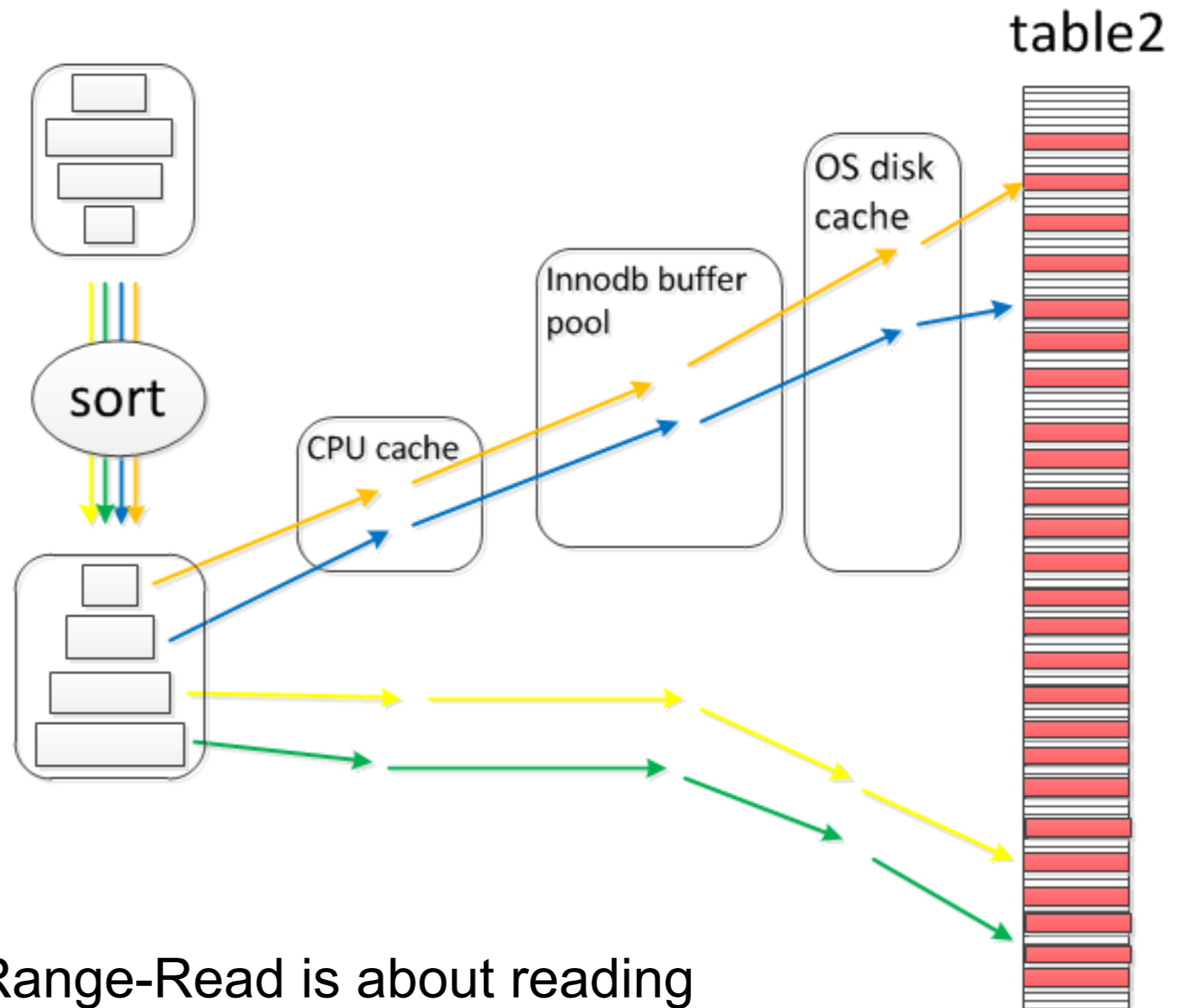


Why are batched index lookups faster?



With BKA:

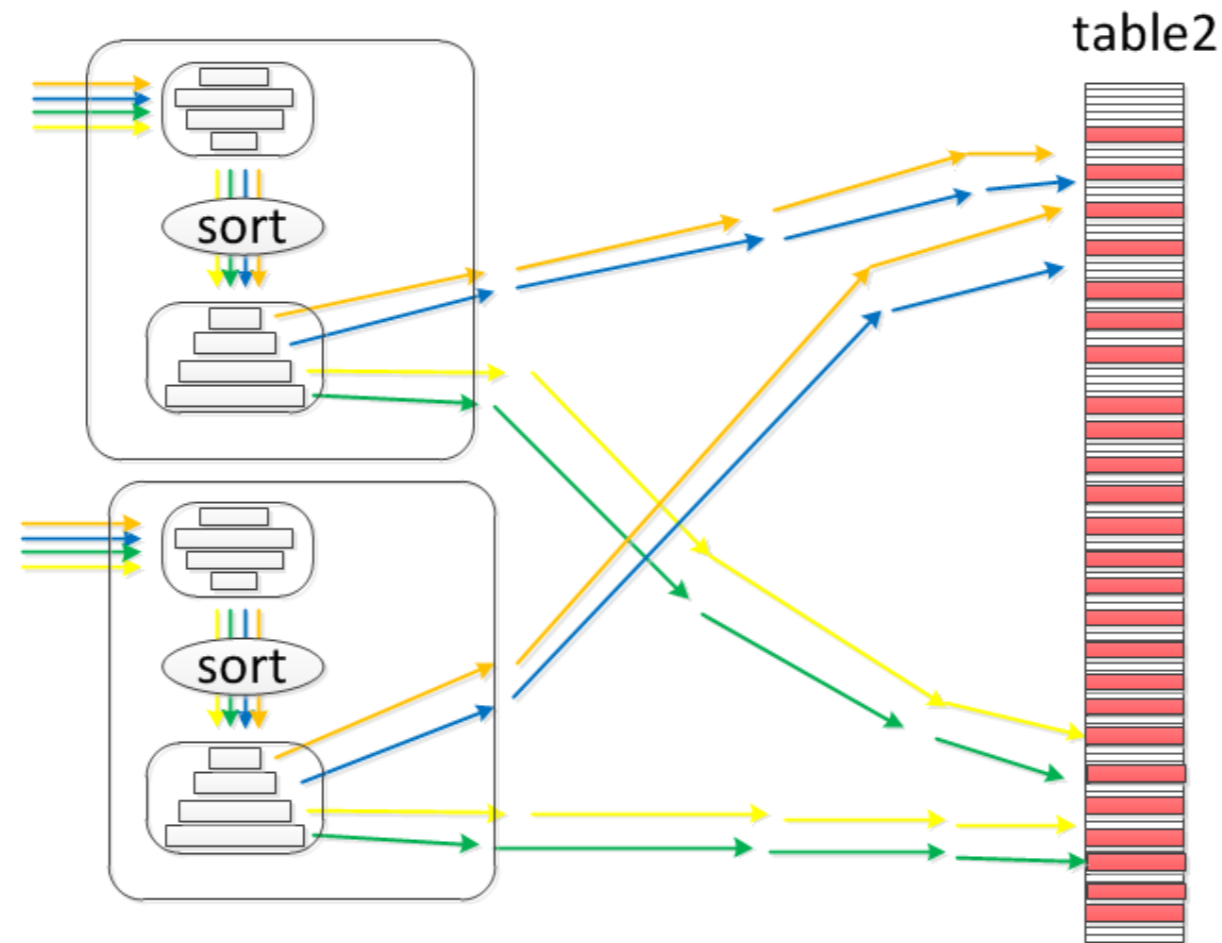
- We sort the buffer
- And then read data in order
- That's cache-friendly
- And prefetch-friendly
- Btw, MySQL 5.6's Multi-Range-Read is about reading in order. But only for range access



Batched Key Access parameters



- What if there is not enough space in the buffer?
- Multiple passes:



- BKA's most important parameter: **`@@join_buffer_size`**



BKA benchmark

BKA Benchmark



```
set join_cache_level=6;
```

```
select max(l_extendedprice)
  from orders, lineitem
where
  l_orderkey=o_orderkey and
  o_orderdate between $DATE1 and $DATE2
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	orders	range	PRIMARY, <u>i_o_orderdate</u>	<u>i_o_orderdate</u>	4	NULL	142680	Using where; Using index
1	SIMPLE	<u>lineitem</u>	ref	PRIMARY	PRIMARY	4	orders. <u>o_orderkey</u>	2	Using join buffer (flat, BKA join); Key-ordered scan

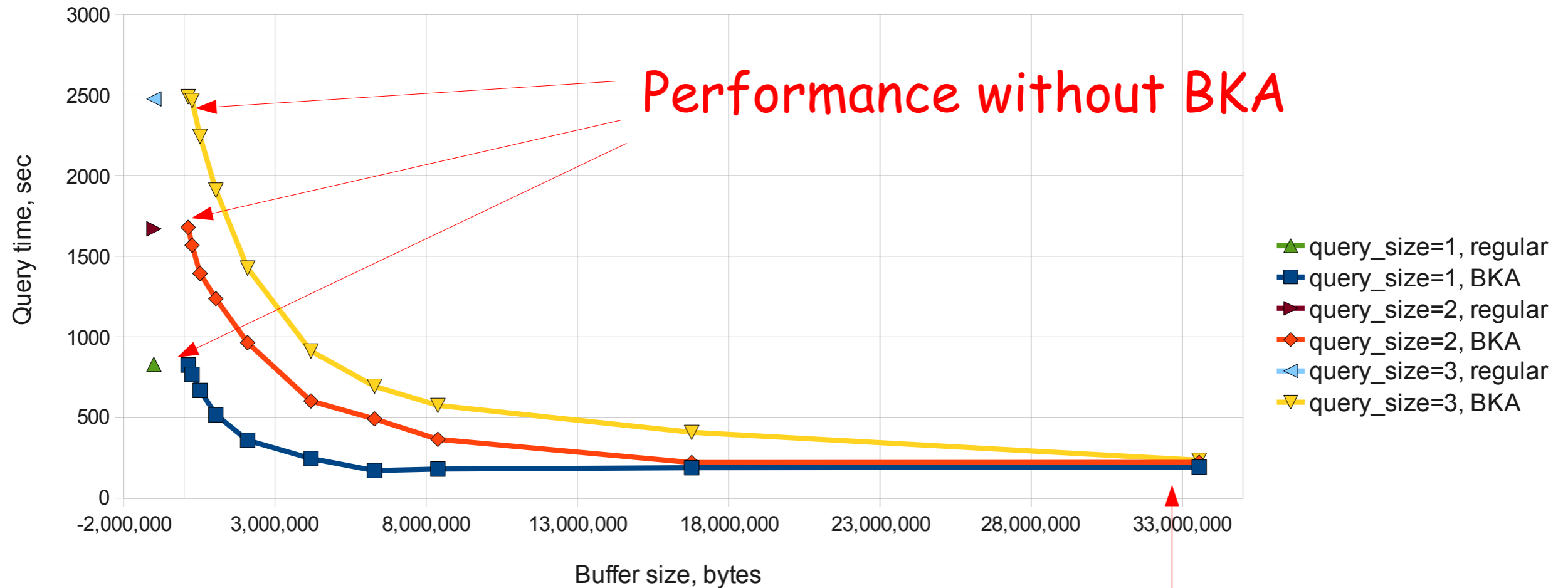
Run with

- Various **join_buffer_size** settings
- Various size of \$DATE1...\$DATE2 range

New Batched Key Access Speedups



BKA join performance depending on buffer size



Performance without BKA

Performance with BKA,
given sufficient buffer size

- Speedups: 4x-10x

Batched Key Access – how to use



- Documentation at <http://kb.askmonty.org>
- Google for “batched key access”
- Basic controls
 - SET join_cache_level=6;
 - SET join_buffer_size=10M..100M
 - SET join_buffer_space_limit= ...
 - per-query limit, i.e. @@join_buffer_size * tables_in_join
 - SET optimizer_switch='join_cache_hashed=off'
 - (disable hash join)
- Then look at EXPLAINS:

type	table	type	possible_keys	key	key_len	ref	rows	Extra
	orders	range	i_o_orderdate	i_o_orderdate	4	NULL	142680	Using where; Using index
	lineitem	ref	i_l_orderkey	i_l_orderkey	4	orders.o_orderkey	2	Using join buffer (flat, BKA join)

New query optimizer features

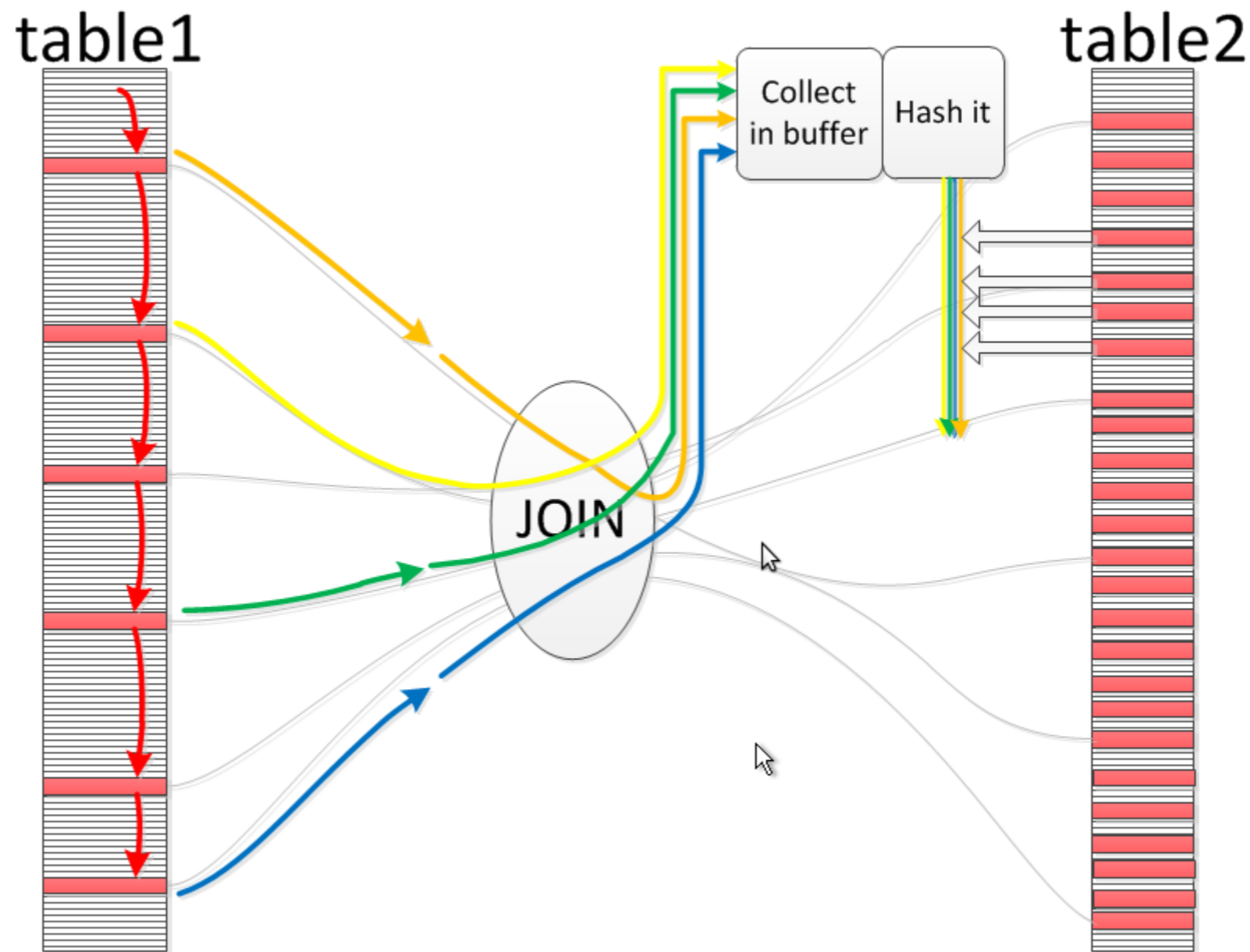


- Correct optimization of index_merge vs range
- index_merge/sort-intersect
- Batched Key Access
- Hash join

Hash join



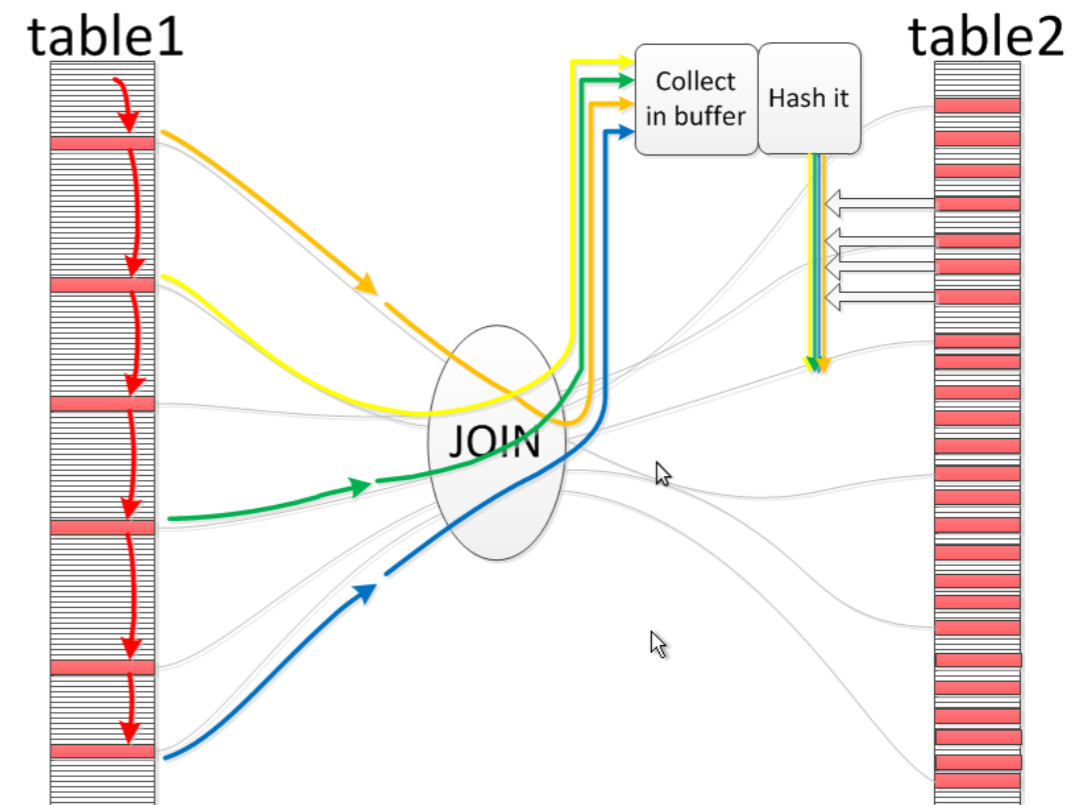
- It's essentially “Using join buffer” with hashed access to the buffer





Targeted use cases

- There are no suitable indexes
- There is a suitable index
- but it's not sufficiently selective for BKA



Hash join benchmark

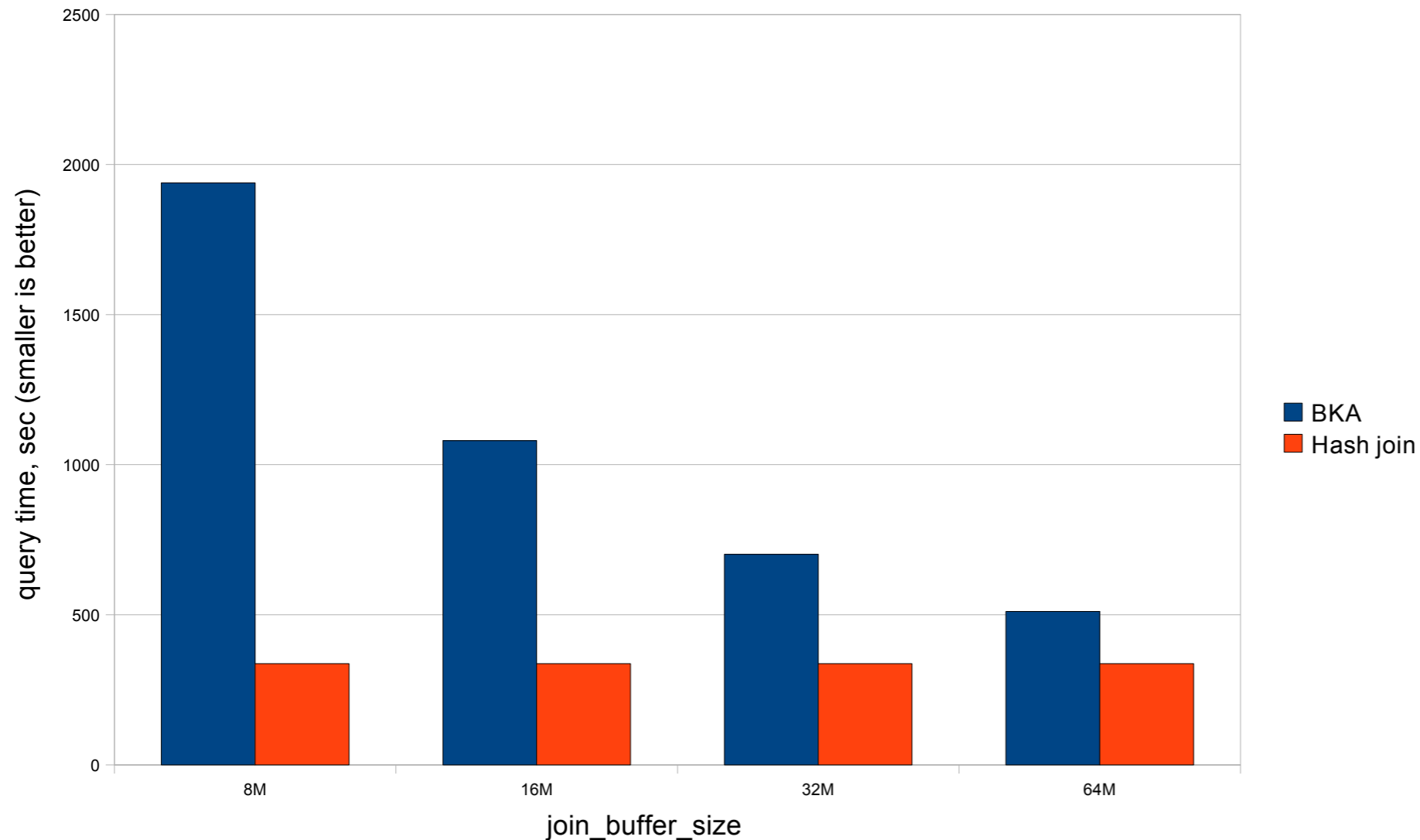


```
MariaDB [dbt3sf10]> explain select max(l_discount) from part, lineitem
  where l_partkey = p_partkey and p_retailprice > 1800\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
      table: part
      type: ALL
possible_keys: PRIMARY
      key: NULL
  key_len: NULL
      ref: NULL
      rows: 1991995
  Extra: Using where
***** 2. row *****
      id: 1
  select_type: SIMPLE
      table: lineitem
  type: hash_ALL
possible_keys: i_1_suppkey_partkey,i_1_partkey
  key: #hash#i_1_suppkey_partkey
  key_len: 5
  ref: dbt3sf10.part.p_partkey
      rows: 60133474
  Extra: Using join buffer (flat, BNLH join)
2 rows in set (0.00 sec)
```

Hash join benchmark



```
select max(l_discount)
from part, lineitem
where
  l_partkey = p_partkey and p_retailprice > 1800;
```



- Speedups:
- 1x-5x wrt BKA
 - Without BKA, the query didn't finish

Hash join summary



- “Classic” (i.e. not “GRACE”)
- Implemented as extension to BKA/join buffering scheme
- No full optimizer support, mostly switches/ parameters so far.
- Execution part is finished and stable
- Optimization is more challenging
 - Hash join is OFF by default
 - Currently mostly controlled by server settings
- To try:
 - SET optimizer_switch='join_cache_hashed=on';
 - SET join_cache_level=4+;
 - SET join_buffer_size=...;

table	type	possible_keys	key	key_len	ref	rows	Extra
orders	range	PRIMARY	i_o_orderdate	4	NULL	142680	Using where; Using index
lineitem	hash	PRIMARY	PRIMARY	4	orders.o_orderkey	60203402	Using join buffer (flat, BNLH join)



- New features
 - Correct optimization of `index_merge` vs `range`
 - `index_merge/sort-intersect`
 - Batched Key Access
 - Hash join
- All are/can be turned off unless we're sure the new feature can't make anything worse
 - => you aren't “burning any bridges” when migrating to newer MariaDB.

Thanks



Q & A