



# State of MariaDB

Igor Babaev  
igor@askmonty.org

# New features in MariaDB 5.2



- New engines: OQGRAPH, SphinxSE
- Virtual columns
- Extended User Statistics
- Segmented MyISAM key cache
- Pluggable Authentication
- Storage-engine-specific CREATE TABLE
- Enhancements to INFORMATION SCHEMA.PLUGINS table
- Group commit for the Aria engine.

# New features in Maria 5.3



- Index Condition Pushdown
- Multi-Range-Read table access (with keys & rowids sorting)
- Block Nested Loop Algorithm for outer joins
- Join algorithms based on Batch Key Access
- Block Nested Loop Hash Join (classic Hash Join)
- Possible materialization for all non-correlated subqueries
- Numerous strategies to optimize subqueries
  - Pull out, First match, Duplicate elimination, In-Out strategies, SQ cache
- Merged derived tables
- Late materialization of derived tables and views
- Index intersect, index merge enhancements
- Faster HANDLER commands
- Microsecond support
- Dynamic columns support.

# Stable new optimizer features in MariaDB 5.3



- Features that are finished (pushed to MariaDB's “trunk”)
  - But not yet released/presented
- The features
  - Correct optimization of index\_merge vs range
  - index\_merge/sort-intersect
  - Batched Key Access improvements
  - Hash join

# Stable new optimizer features in MariaDB 5.3



- **Correct optimization of index\_merge vs range**
- index\_merge/sort-intersect
- Batched Key Access
- Hash join

# index\_merge vs range optimization



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

```
MySQL [ontime]> select count(*) from ontime;
```

```
+-----+
|count(*)|
+-----+
| 1578171|
+-----+
```

```
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|
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

```
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|
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

- => 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 (and so, next major MariaDB release)
- New functionality is always on (no way to switch to old behavior)
- No known problems or gotchas

# Stable optimizer features in MariaDB 5.3

- 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\*:

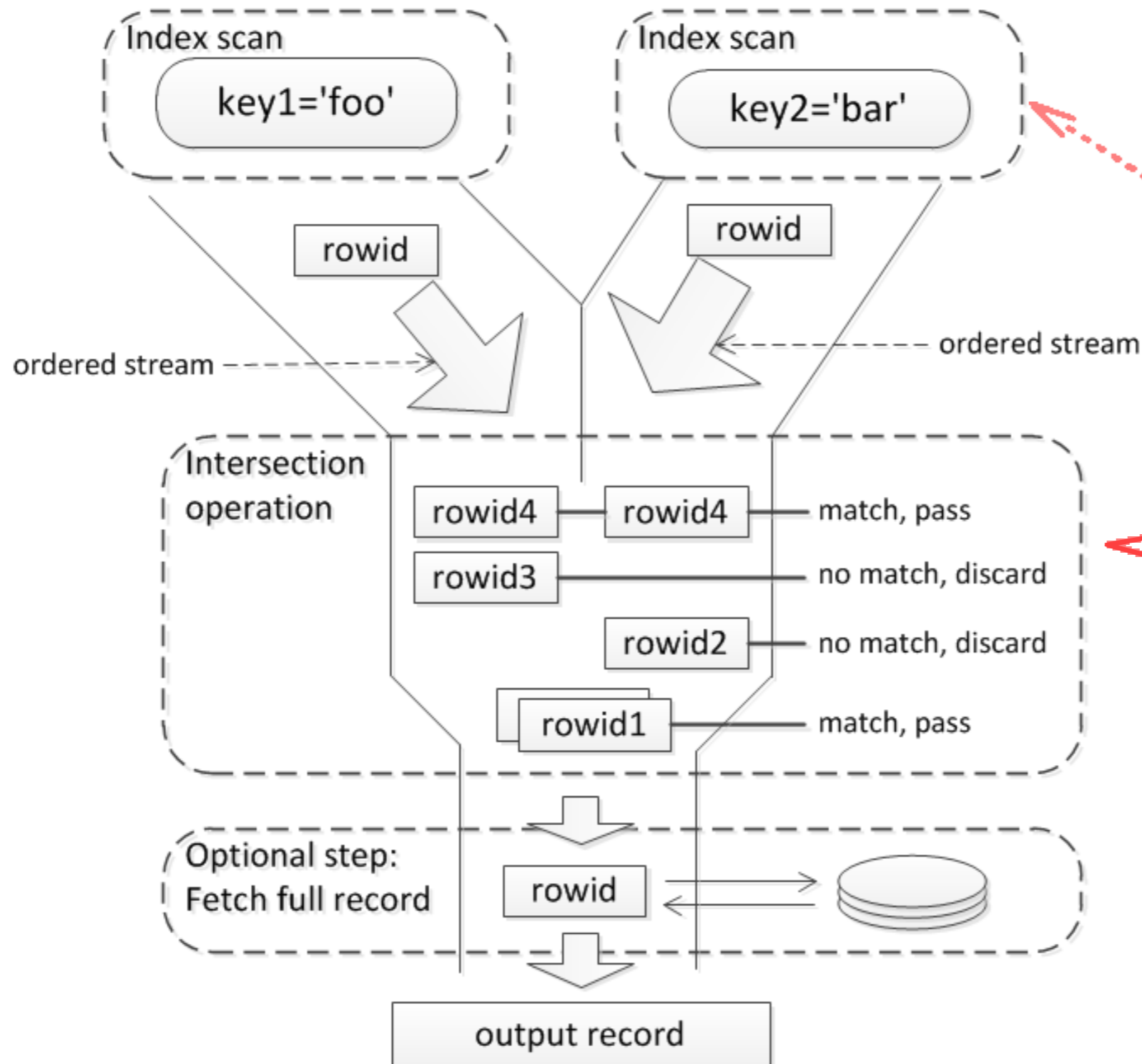
```
MySQL [ontime]> explain select avg(arrdelay) from ontime where depdel15=1 and OriginState IN ('CA', 'GB');
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|id|select_type|table |type|possible_keys      |key              |key_len|ref  |rows |Extra
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+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

Just so we have a non-equality here

MariaDB 5.3 doesn't have this limitation:

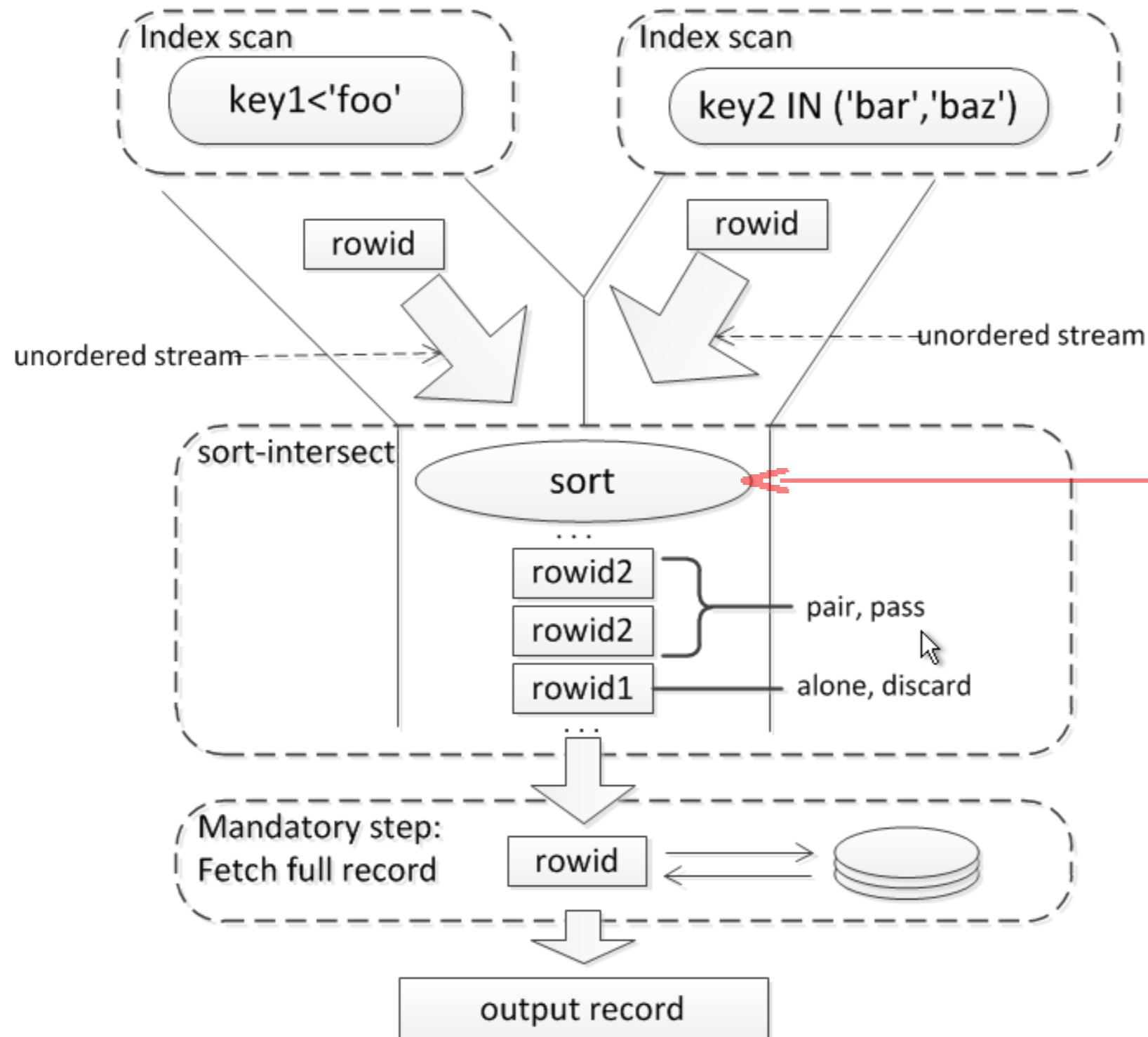
```
MariaDB [ontime]> explain select avg(arrdelay) from ontime where depdel15=1 and OriginState IN ('CA', 'GB');
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|id|select_type|table |type      |possible_keys      |key              |key_len|ref  |rows |Extra
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 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
```

# index\_merge/intersect inside



This module relies heavily on source streams being ordered ... which means we can have only equalities there

# index\_merge/sort\_intersect inside



sort\_intersect takes care of ordering on his own.

# Speedups provided by sort-intersect



A couple of attempts (1.5 M rows table, “hot” buffer pool):

```
select avg(arrdelay) from ontime where depdel15=1 and OriginState IN ('CA', 'GB');
```

|                   |               |
|-------------------|---------------|
| Query time before | 10.78         |
| Query time after  | 3.23          |
| Improvement:      | <b>3.34 x</b> |

```
select count(*) from ontime where depdelay > 30 and OriginState IN ('WA', 'CA');
```

|                   |               |
|-------------------|---------------|
| Query time before | 18.27         |
| Query time after  | 2.37          |
| Improvement:      | <b>7.71 x</b> |

Conclusions after running some similar queries:

- improvement is greater when \*both\* ranges are sufficiently big
- Performance seems to depend on factors not fully accounted for by the optimizer

# Possible sort\_intersect gotchas

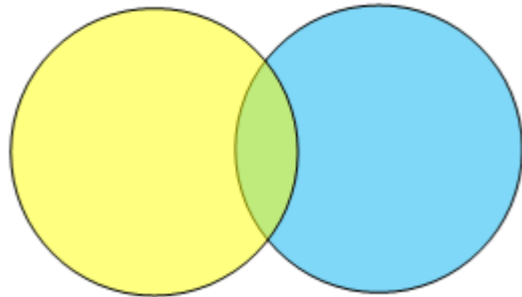


## Problem #1: correlated data

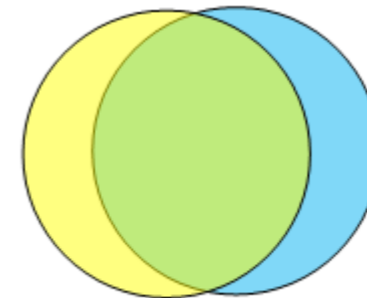
```
select ... from ontime
  where depdelay>30 and < Airline='Punctual Fliers'
                        Airline='MessyAir'
```

Index intersection reduces number of records we'll read...

'Punctual Fliers'

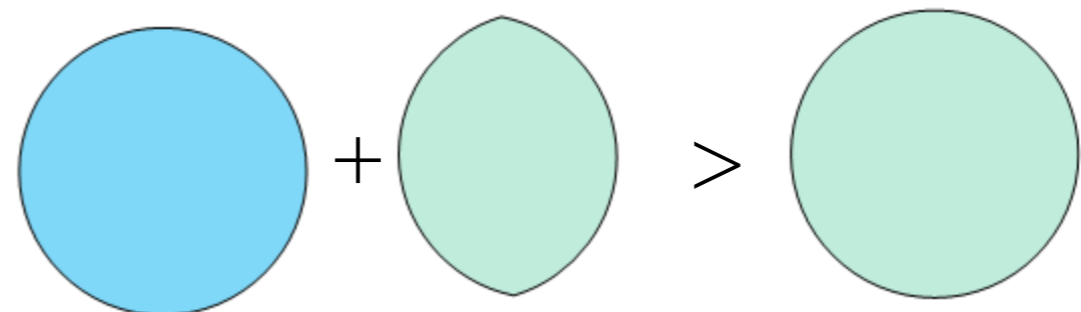
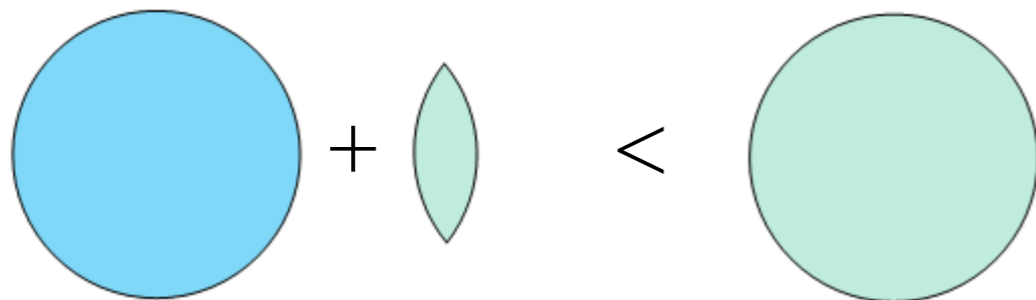


'MessyAir'



... but at the cost of adding scan on the second index.

Is it worth it? Sometimes.



# Possible `sort_intersect` gotchas



- Problem #1: correlated data
  - No way to know from [index] statistics
- Problem #2: un-tuned cost model.

=> `sort_intersect` is disabled by default.

- If you
  - Have tables with lots of records
  - Look for intersections of sufficiently big (but still small enough to make table scan an overkill) populations

**SET optimizer\_switch='index\_merge\_intersection=on'**

... and let us know how it worked.

# Stable new optimizer features in MariaDB 5.3



- Correct optimization of index\_merge vs range
- index\_merge/sort-intersect
- **Batched Key Access**
- Hash join



- Batched Key Access feature (“BKA”)
  - Backported to MariaDB 5.3
  - A number of related bugs (“MRR bug pile”) fixed
  - New improvement “Key ordered retrieval”
- Hash Join
  - “Classic” (i.e. not “grace”) implemented as extension to BKA/join buffering scheme
  - No full optimizer support, mostly switches/parameters so far.



# BKA and Hash Join: background



table1

A vertical grid representing a table with 32 rows and 3 columns. The grid is divided into 8 groups of 4 rows each. Each group is separated by a thick red horizontal bar. The first group is at the top, followed by a red bar, then the second group, another red bar, and so on, ending with a red bar at the bottom.

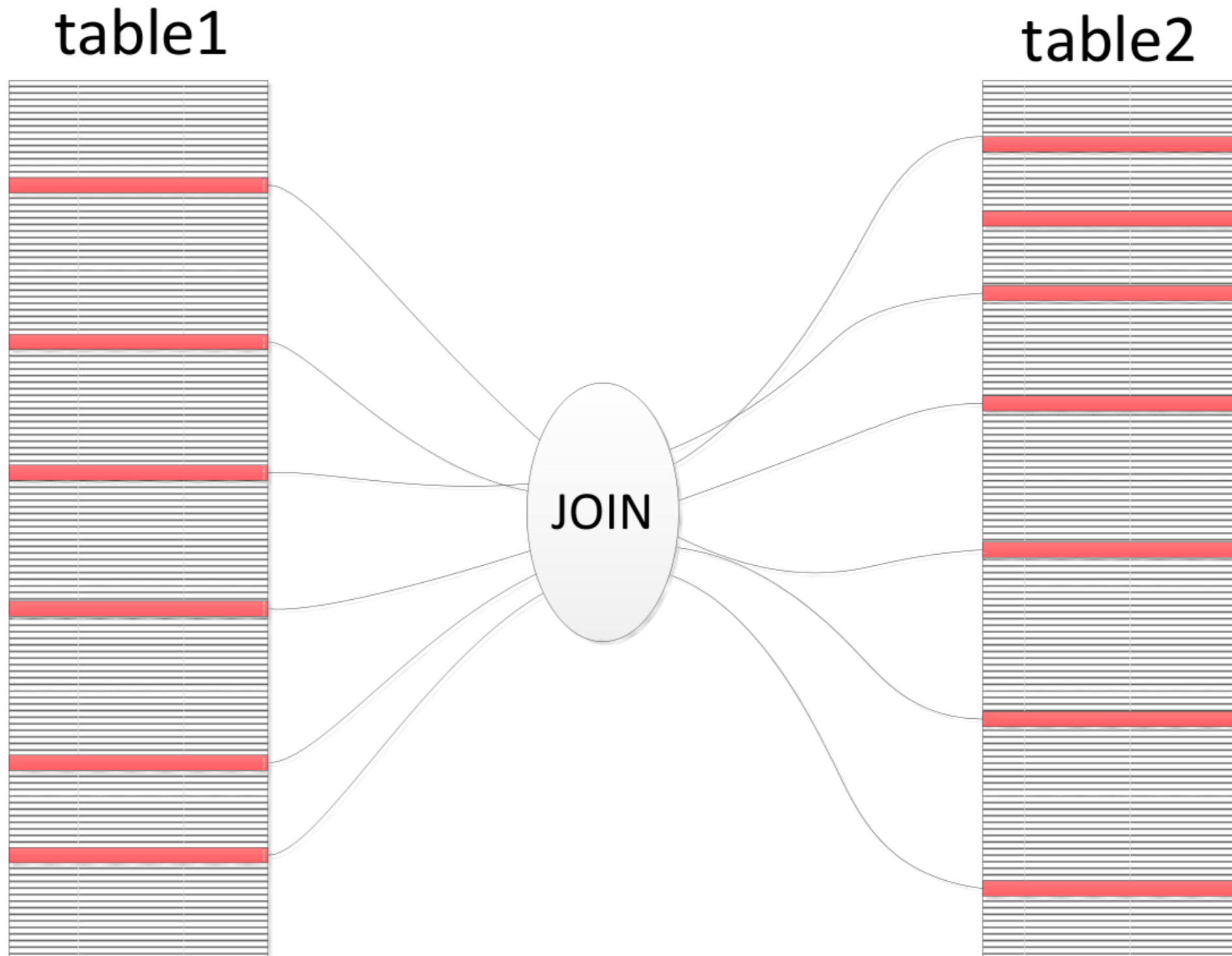
table2

A vertical grid representing a table with 32 rows and 3 columns. The grid is divided into 8 groups of 4 rows each. Each group is separated by a thick red horizontal bar. The first group is at the top, followed by a red bar, then the second group, another red bar, and so on, ending with a red bar at the bottom.

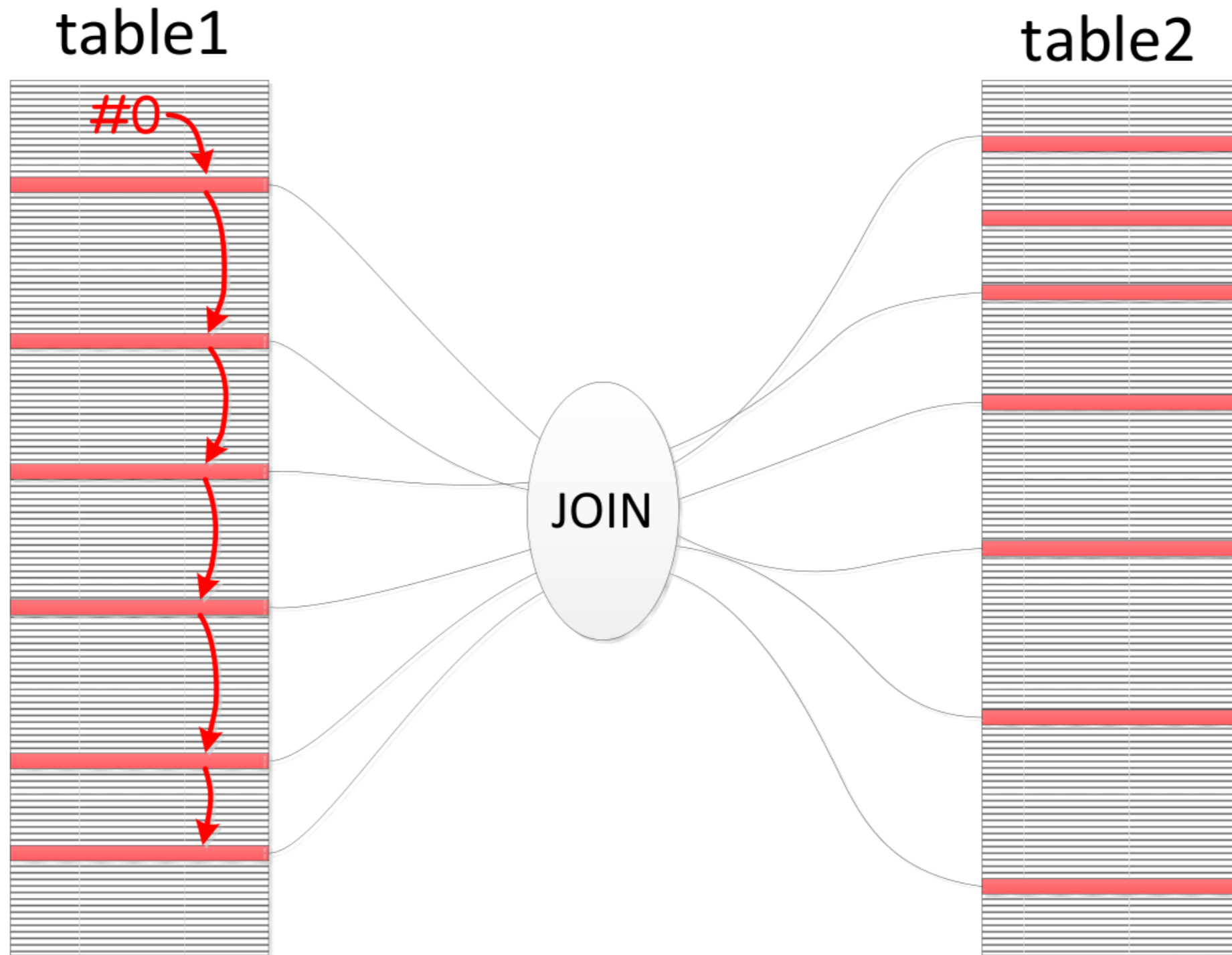
# BKA and Hash Join: background



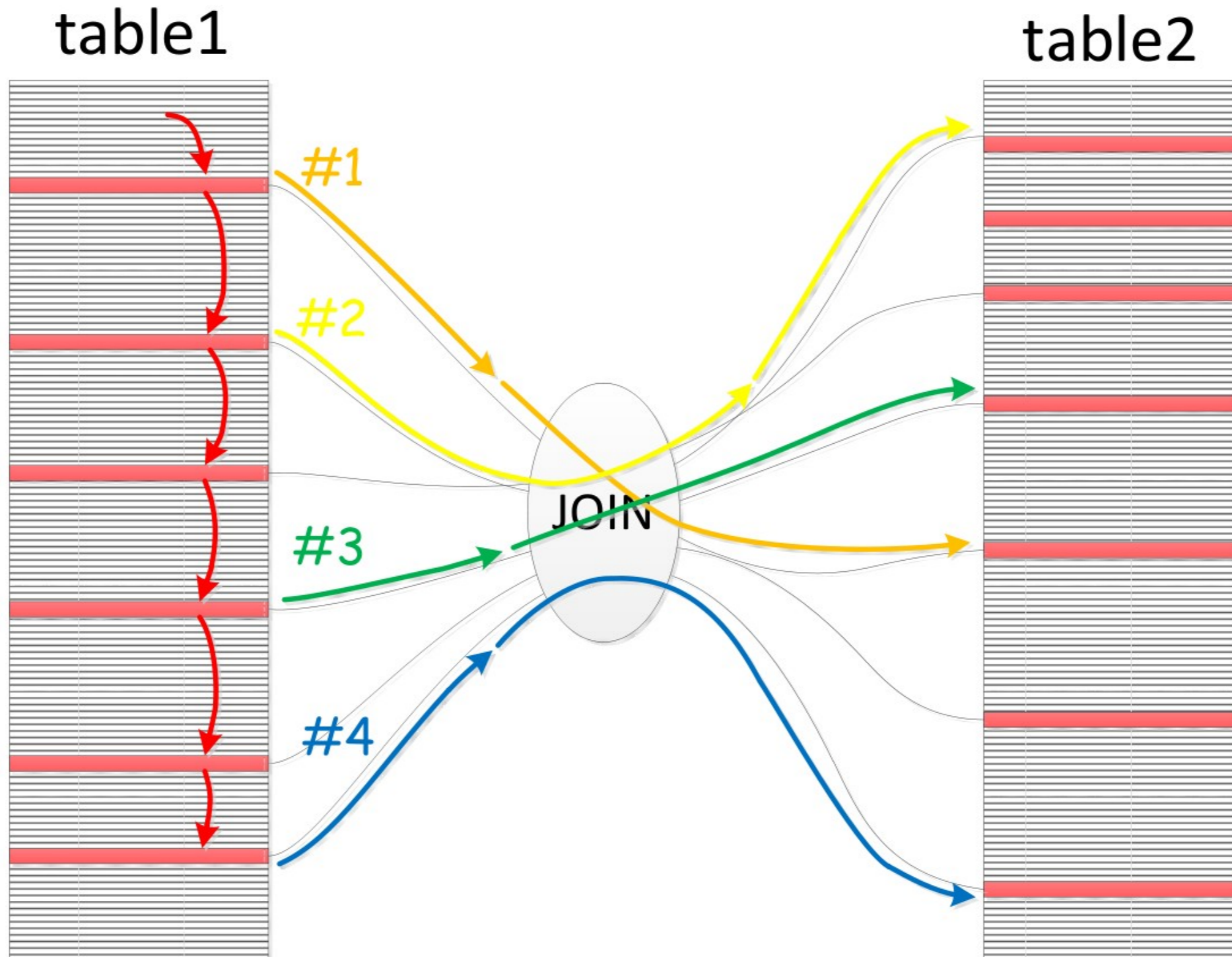
# BKA and Hash Join: background



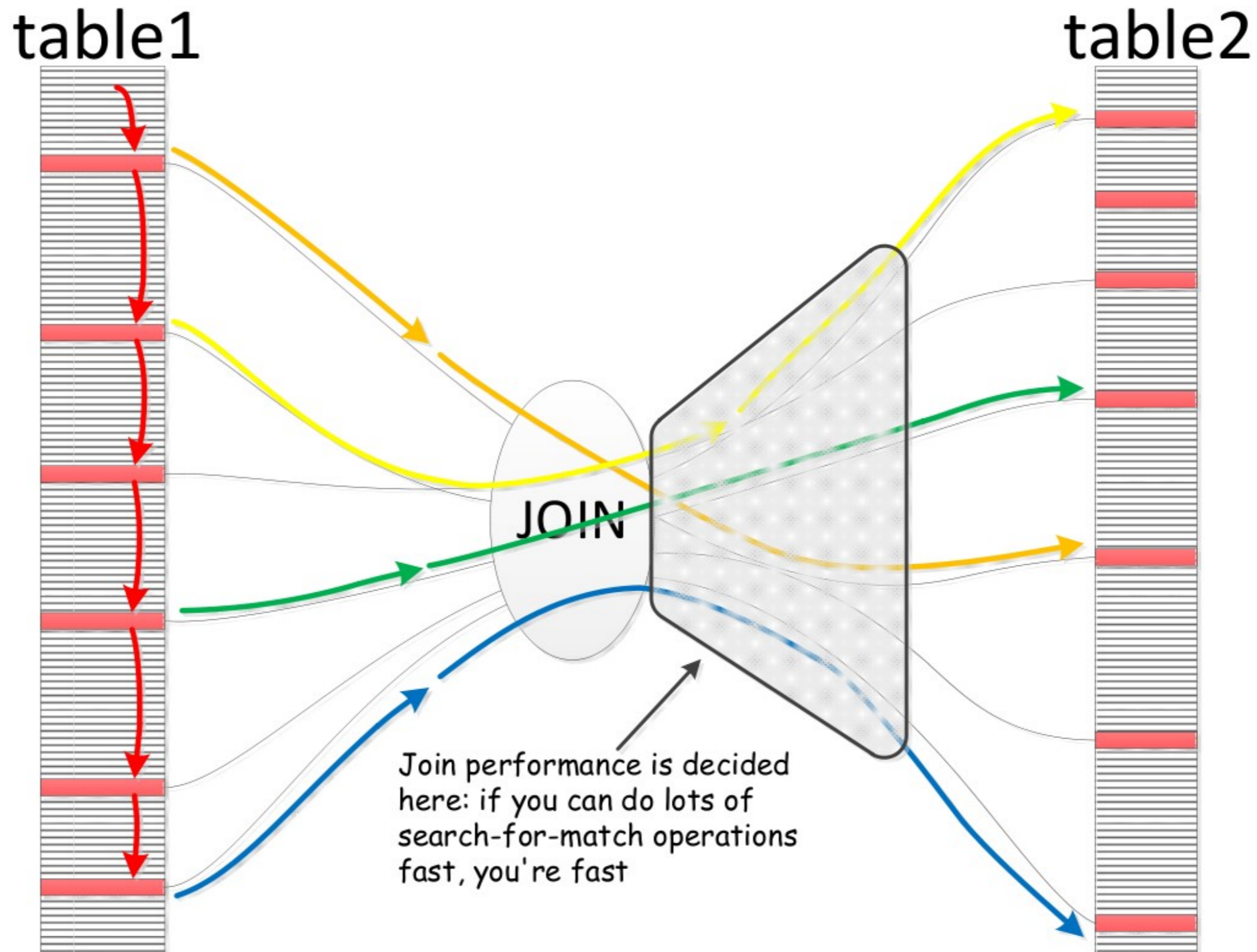
# BKA and Hash Join: background

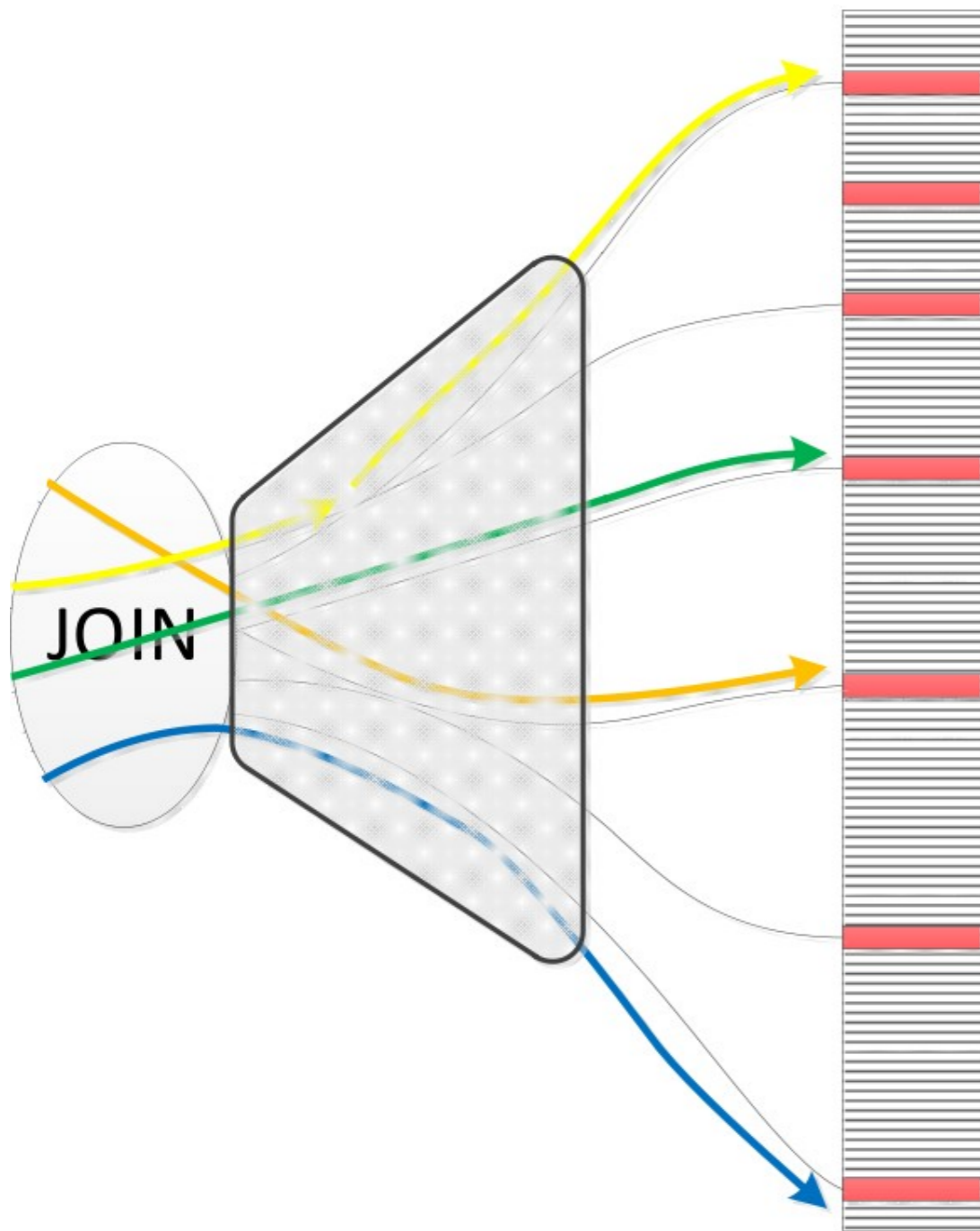


# BKA and Hash Join: background



# BKA and Hash Join: background





## General ideas

1. Make each individual *search-for-match* operation fast
2. Search for matches for whole groups, and make *group* search faster

## Implementations in MySQL

- #1, *ref access*:  
Ask the DBA to create indexes, let the optimizer use them for lookups
- #2: “*Using join buffer*”
  - Put lookup requests into buffer;
  - Scan the table, checking each record against each request in the buffer

# Batched Key Access



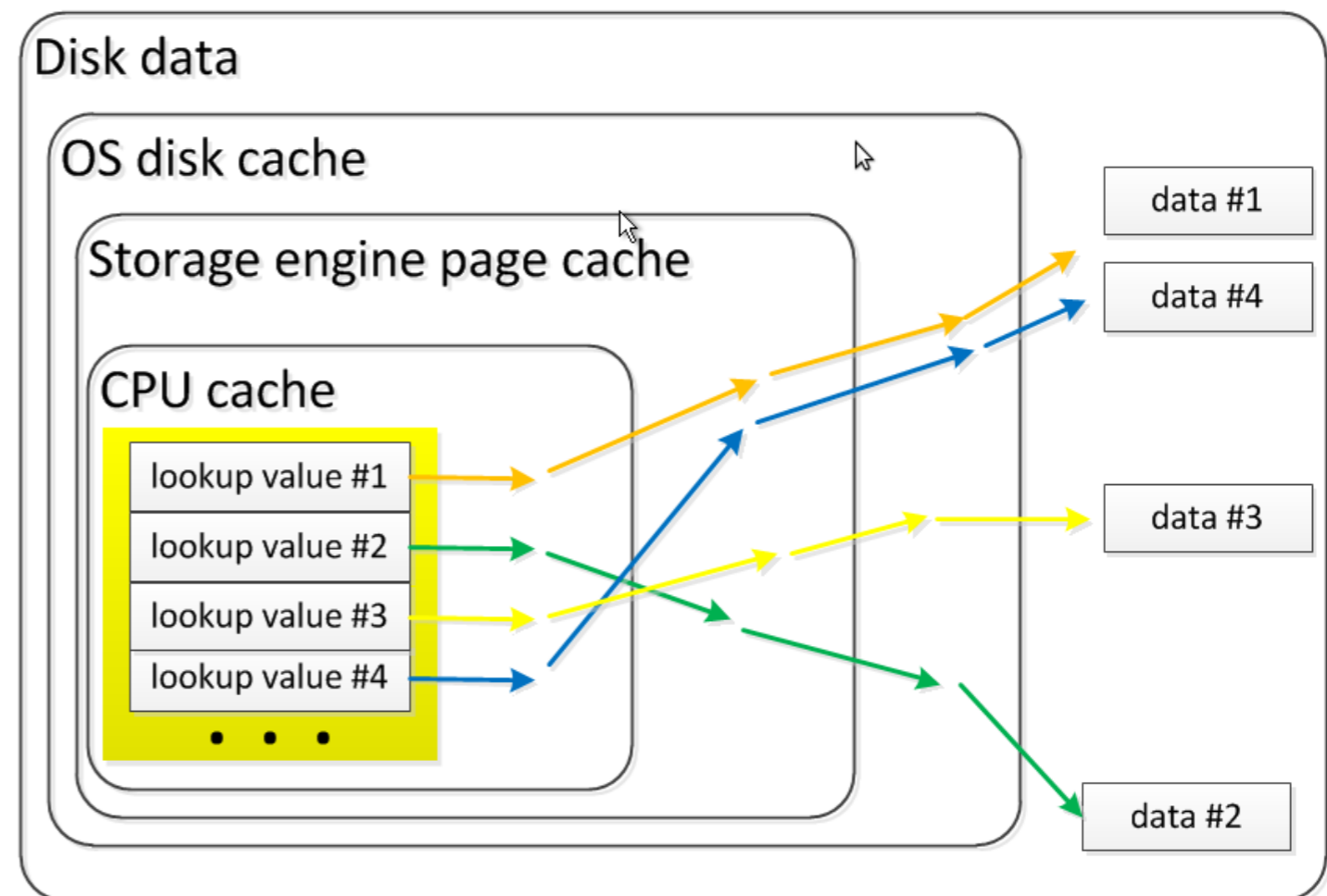
Batched Key Access uses both #1 and #2. It is essentially a *group-index-lookup*:

- it uses an index, just like ref access
- but it makes lookups for groups of requests

Q: Why making index lookups in groups would be faster?

A1: caching: reordering lookup requests so that “close” data is accessed together will improve cache hit ratio

A2: if you make requests in order, disk/mem pre-fetch will provide speedup.

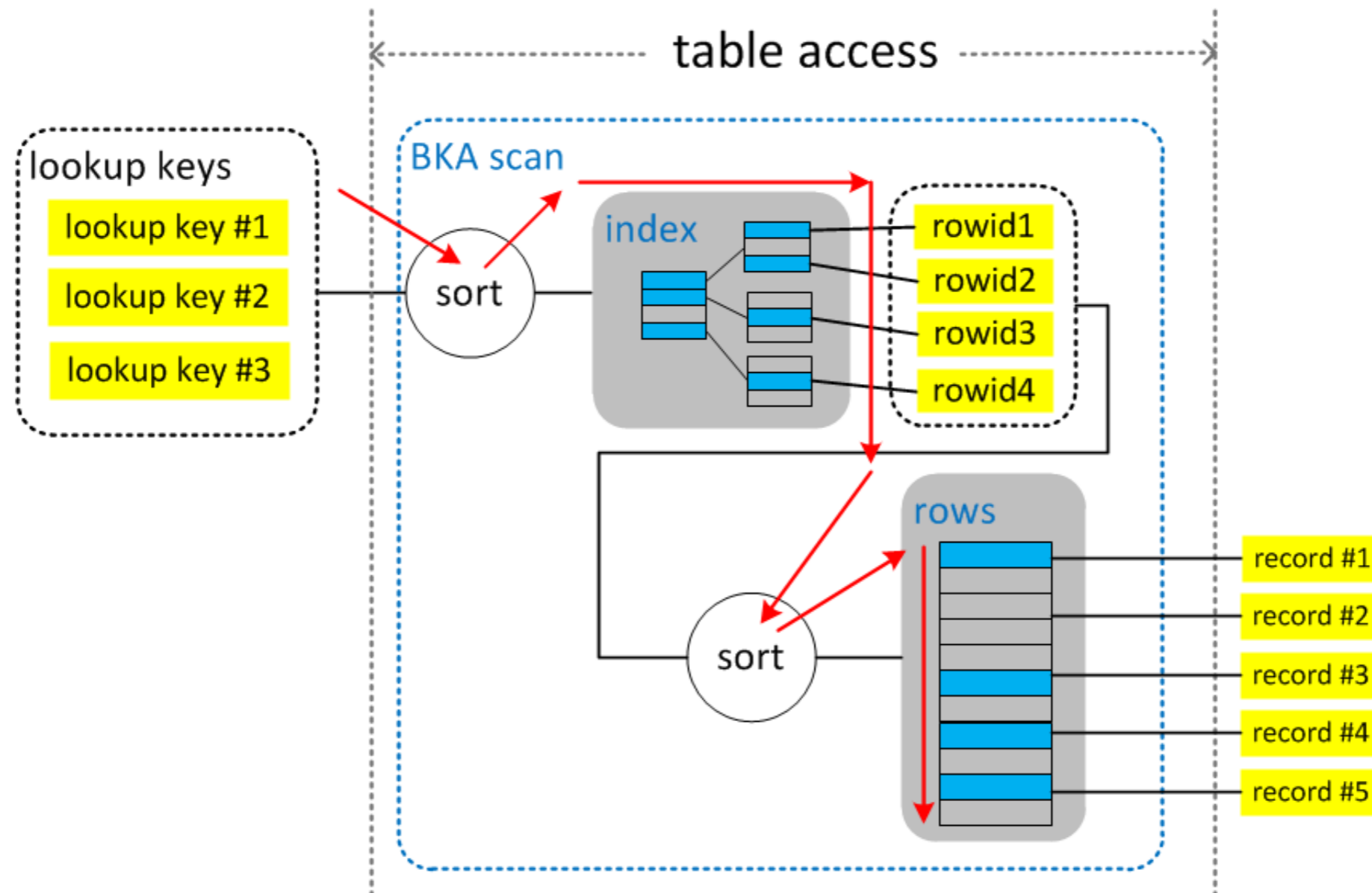


# Batched Key Access – table level



BKA achieves speedups by having storage engine (=table) to

- Sort rowids before reading table rows
- Sort key values before accessing the index (new!)



# Batched Key Access - speedups



So when is BKA practically useful?

- Big database
  - e.g. DBT-3 sf=10, 30G, biggest table 60M rows
    - innodb\_buffer\_pool=4G (1/10th): speedups for queries over “cold” data
    - innodb\_buffer\_pool=256 (1/100th): speedups for all queries
- Big joins
  - Make 100K or more lookups (check EXPLAIN)
- Observed speedups:
  - Accessed data mostly in buffer pool: ~ 3x,
  - Some of accessed data not in buffer pool: 10x
  - Significant portion is read from disk: 50x-100x

# Batched Key Access – how to use



- Some 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 just in case)
- 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) |

# Stable new optimizer features in Maria DB 5.3



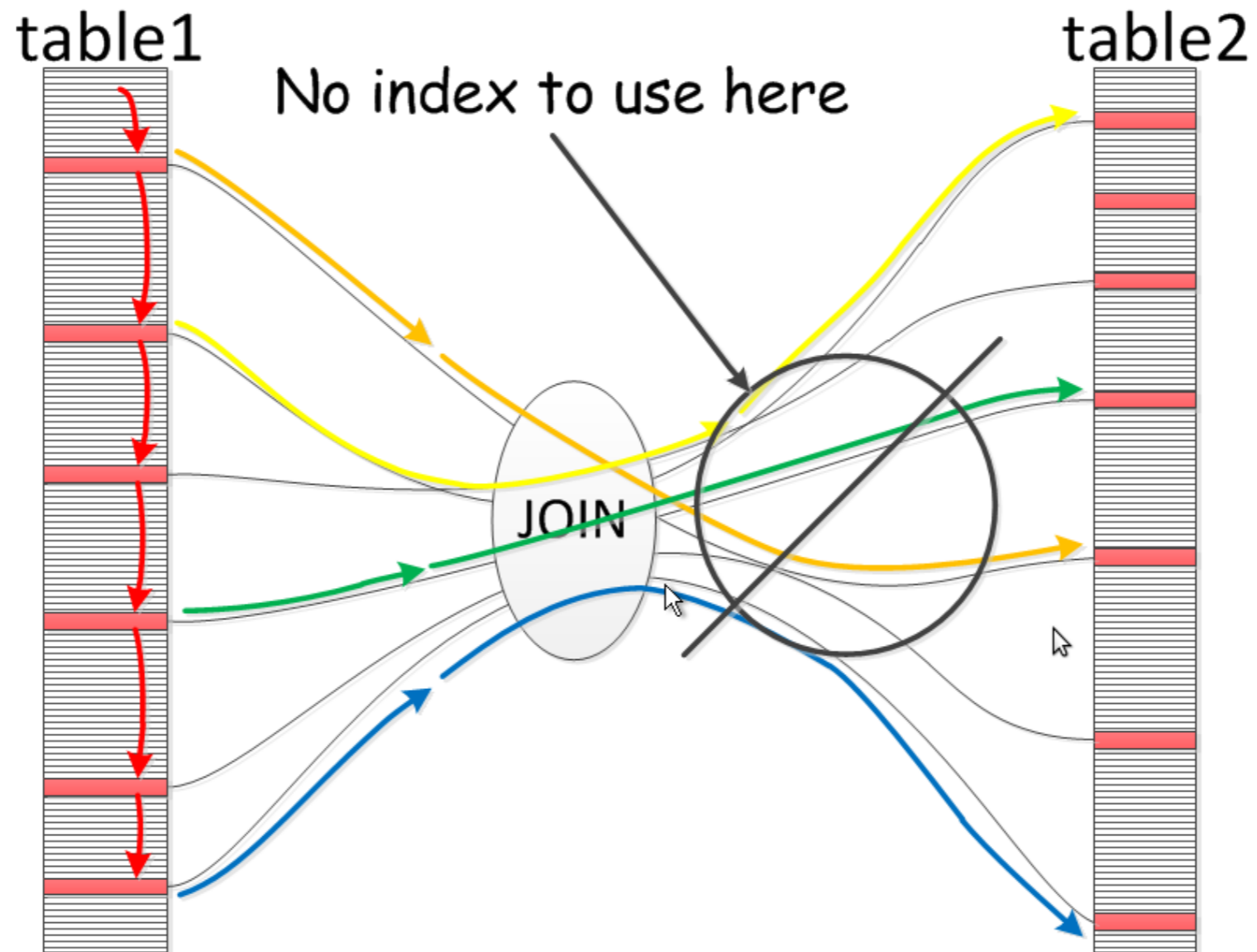
- Correct optimization of index\_merge vs range
- index\_merge/sort-intersect
- Batched Key Access
- Hash join

# Hash join



BKA is not the answer for all big-join needs.

1. It can't be used when there are no suitable indexes

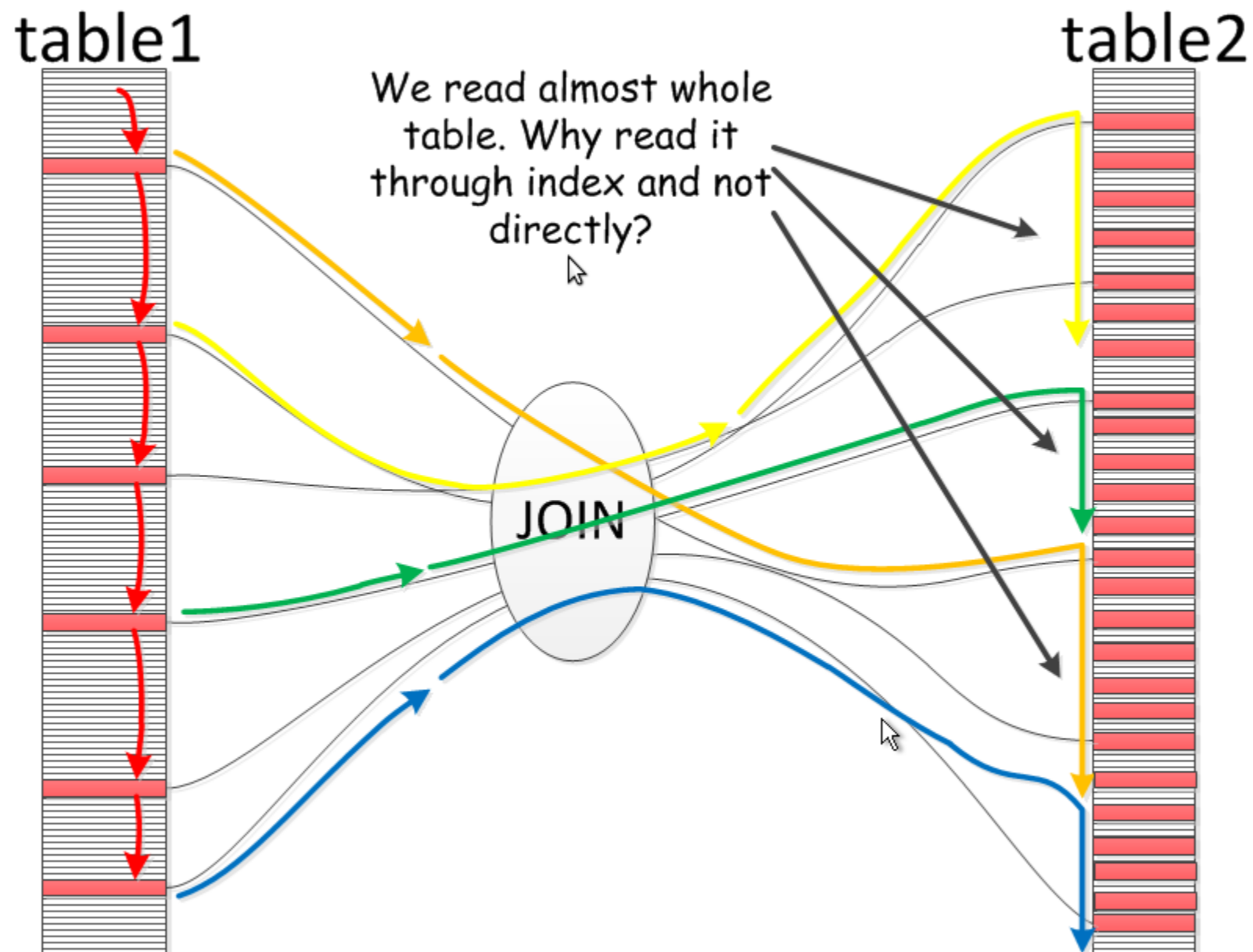


# Hash join



BKA is not the answer for all big-join needs

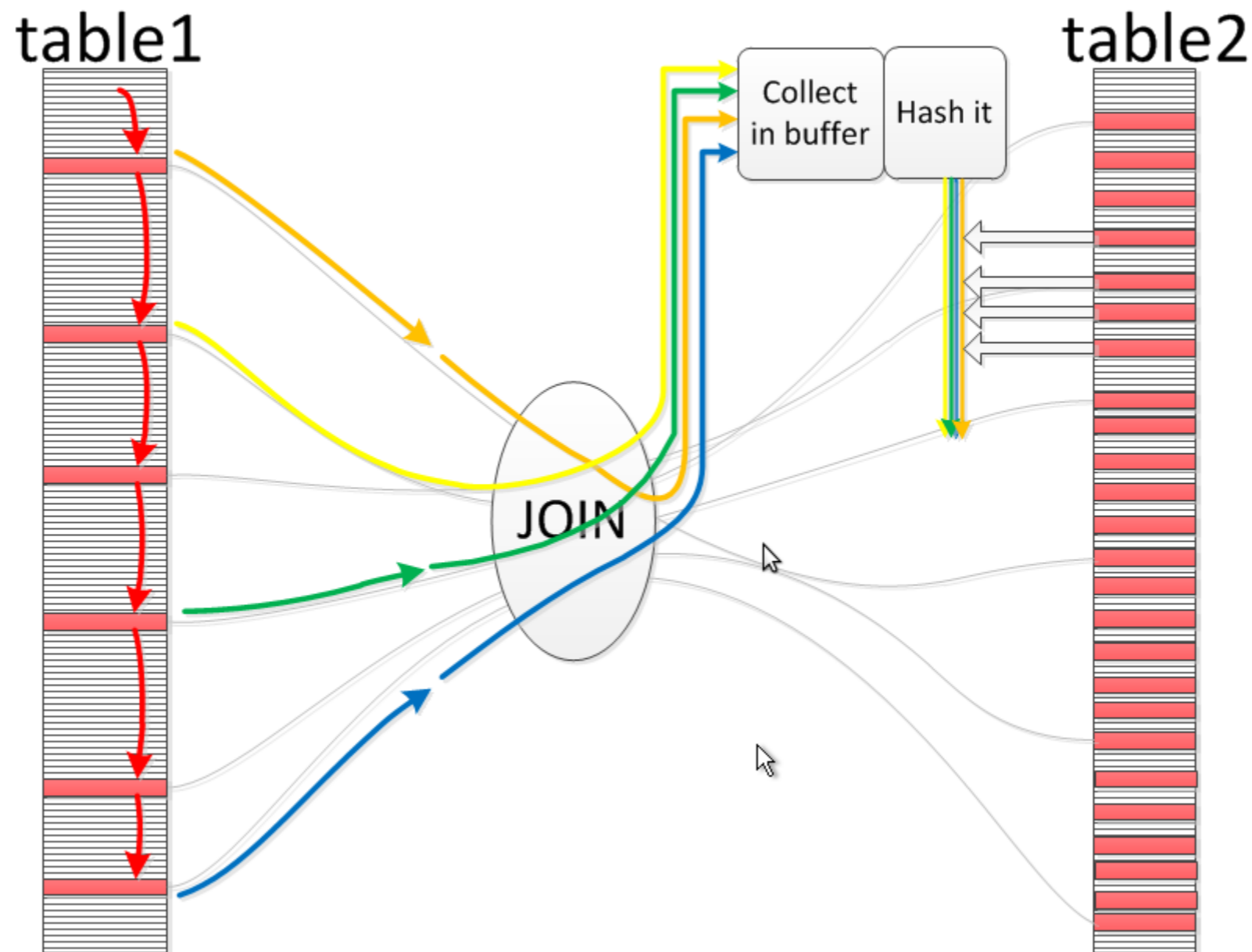
2. It only brings overhead when there are lots of matches



# Hash join



- Answer to BKA problems: HASH join
  - It's essentially "Using join buffer" with hashed access to the buffer



# Hash join summary



- Covers BKA's weak spots in big-join handling
- Execution part is finished and presumed 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