ADVANCED DATABASE SYSTEMS

Multi-Way Join Algorithms

10

Andy Pavlo
CMU 15-721
Spring 2024

Carnegie Mellon University

LAST CLASS

Different methods to execute a parallel hash join.

No partitioning hash join is good enough to get started with minimal engineering optimizations.

MOTIVATION

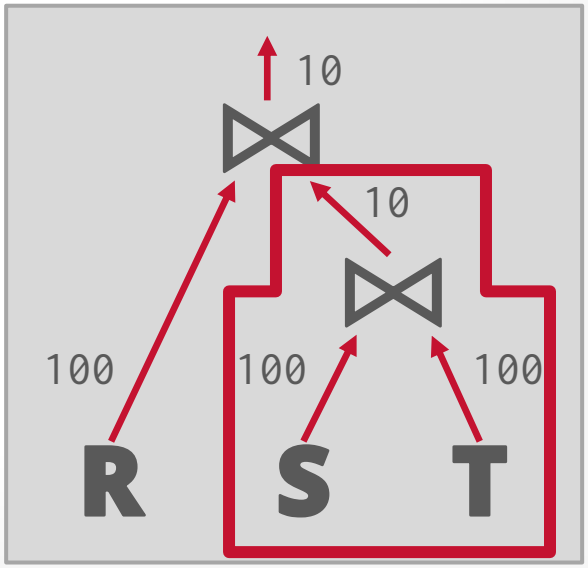
Binary (i.e., two table) joins are ubiquitous in relational DBMSs.

→ Decades of research to make these algorithms highly efficient.

This is the optimal approach when the output of the join is smaller than its two inputs.

But things go bad when a join's output is larger than its inputs...

```
SELECT *
  FROM R, S, T
 WHERE R.a = S.a
   AND R.b = T.a
   AND S.b = T.b
```



MOTIVATION

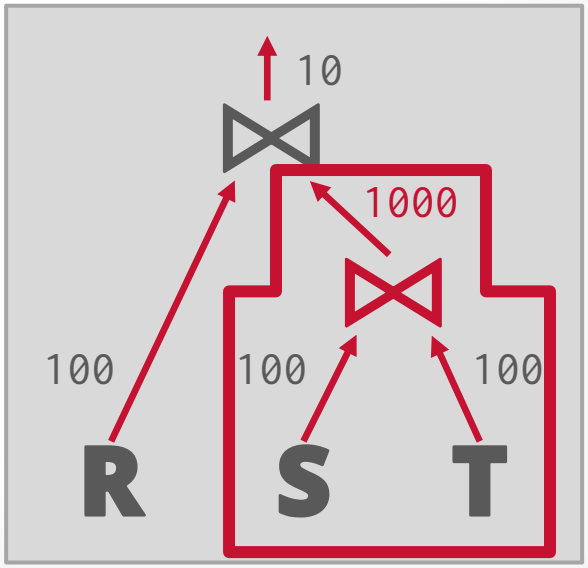
Binary (i.e., two table) joins are ubiquitous in relational DBMSs.

→ Decades of research to make these algorithms highly efficient.

This is the optimal approach when the output of the join is smaller than its two inputs.

But things go bad when a join's output is larger than its inputs...

```
SELECT *
  FROM R, S, T
 WHERE R.a = S.a
   AND R.b = T.a
   AND S.b = T.b
```



MOTIVATION

```
SELECT *
  FROM R, S, T
 WHERE R.a = S.a
   AND R.b = T.a
   AND S.b = T.b
```

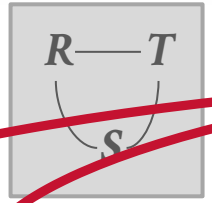


Table R

a	b
0	0
0	1
0	2
1	0
2	0

Table S

a	b
0	0
0	1
0	2
1	0
2	0

Table T

a	b
0	0
0	1
0	2
1	0
2	0

$R \bowtie S$

a	b	c
0	0	0
0	1	1
0	0	2
1	0	0
1	0	1
1	0	2
2	0	0
2	0	1
2	0	2
0	1	0
0	2	0

$R \bowtie S \bowtie T$

a	b	c
0	0	0
0	1	1
0	2	2
1	0	0
2	0	0
0	0	0
0	1	1

MOTIVATION

```
SELECT *
  FROM R, S, T
 WHERE R.a = S.a
   AND R.b = T.a
   AND S.b = T.b
```

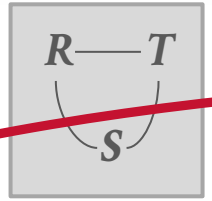


Table R

a	b
0	0
0	1
0	2
1	0
2	0

Table S

a	b
0	0
0	1
0	2
1	0
2	0

Table T

a	b
0	0
0	1
0	2
1	0
2	0

$R \bowtie T$

a	b	c
0	0	0
0	0	1
0	0	2
0	1	0
0	1	1
0	1	2
0	2	0
0	2	1
0	2	2
1	0	0
2	0	0

$R \bowtie T \bowtie S$

a	b	c
0	0	0
0	0	1
0	0	2
0	1	0
0	2	0
1	0	0
2	0	0

MOTIVATION

```
SELECT *
  FROM R, S, T
 WHERE R.a = S.a
   AND R.b = T.a
   AND S.b = T.b
```

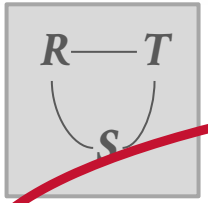


Table R

a	b
0	0
0	1
0	2
1	0
2	0

Table S

a	b
0	0
0	1
0	2
1	0
2	0

Table T

a	b
0	0
0	1
0	2
1	0
2	0

$S \bowtie T$

a	b	c
0	0	0
1	0	0
2	0	0
0	1	0
1	1	0
2	1	0
0	2	0
1	2	0
2	2	0
0	0	1
0	0	2

$S \bowtie T \bowtie R$

a	b	c
0	0	0
0	0	1
0	0	2
0	1	0
0	2	0
1	0	0
2	0	0

MOTIVATION

```
SELECT *
  FROM R, S, T
 WHERE R.a = S.a
   AND R.b = T.a
   AND S.b = T.b
```

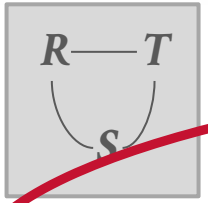


Table R

a	b
0	0
0	1
0	2
1	0
2	0

Table S

a	b
0	0
0	1
0	2
1	0
2	0

Table T

a	b
0	0
0	1
0	2
1	0
2	0

$S \bowtie T$

a	b	c
0	0	0
1	0	0
2	0	0
0	1	0
1	1	0
2	1	0
0	2	0
1	2	0
2	2	0
0	0	1
0	0	2

$S \bowtie T \bowtie R$

a	b	c
0	0	0
0	0	1
0	0	2
0	1	0
0	2	0
1	0	0
2	0	0

Wasted Computation
Wasted Storage

TODAY'S AGENDA

Worst-Case Optimal Joins

Leap-Frog Trie Join

Hash Trie Join

Additional Optimizations

System Profiling + Hardware Counters

WORST-CASE OPTIMAL JOINS

Perform join by examining a variable at a time instead of a relation at a time.

→ First considered by other Germans in 2008.
→ Early implementations were EmptyHeaded and LogicBlox.

The runtime of a WCOJ algorithm is bounded by the output size of the result and depends on the variable evaluation ordering.

→ If more tuples match in the intermediate results, then the DBMS must check the other tables.
→ The more tables a WCOJ algorithm joins, the better its performance relative to the input.

WORST-CASE OPTIMAL JOINS

The worst-case runtime of the algorithm meets a known lower bound for the worst-case runtime of any join algorithm.

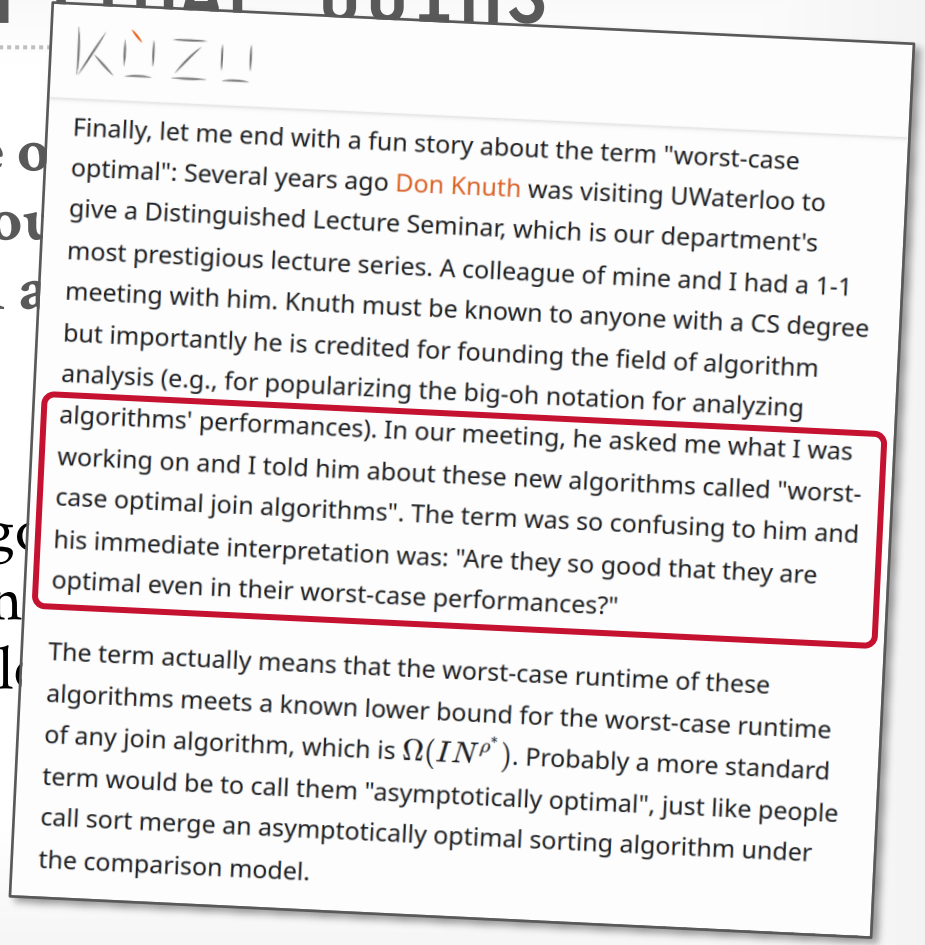
Alternative Definition:
The runtime of the join algorithm is better than all other join algorithms when the query and data represent the worst possible scenario.

Source: Semih Salihoğlu

WORST-CASE OPTIMAL JOINS

The worst-case runtime o
meets a known lower bou
case runtime of any join a

Alternative Definition:
The runtime of the join algo
other join algorithms when
represent the worst possibl



KÙZU

Finally, let me end with a fun story about the term "worst-case optimal": Several years ago Don Knuth was visiting UWaterloo to give a Distinguished Lecture Seminar, which is our department's most prestigious lecture series. A colleague of mine and I had a 1-1 meeting with him. Knuth must be known to anyone with a CS degree but importantly he is credited for founding the field of algorithm analysis (e.g., for popularizing the big-oh notation for analyzing algorithms' performances). In our meeting, he asked me what I was working on and I told him about these new algorithms called "worst-case optimal join algorithms". The term was so confusing to him and his immediate interpretation was: "Are they so good that they are optimal even in their worst-case performances?"

The term actually means that the worst-case runtime of these algorithms meets a known lower bound for the worst-case runtime of any join algorithm, which is $\Omega(IN^{\rho^*})$. Probably a more standard term would be to call them "asymptotically optimal", just like people call sort merge an asymptotically optimal sorting algorithm under the comparison model.

Source: Semih Salihoğlu

WORST-CASE OPTIMAL JOINS

As of 2024, very few DBMSs support worst-case optimal join algorithms.

→ First known implementations were in LogicBlox and EmptyHeaded (Stanford)

These joins will be more common because the SQL 2023 standard includes property graph query extensions (SQL/PGQ).

RelationalAI

UMBRA

LogicBlox

KUZU

SKEW STRIKES BACK: NEW DEVELOPMENTS IN THE THEORY OF JOIN ALGORITHMS
SIGMOD RECORD 2013

LEAP-FROG TRIE JOIN

Relations must be indexes (or sorted) on the join keys in advance before performing the join.

Represent relations with multiple attributes as tries.

→ One trie per relation.

→ Each level in the trie represents a single attribute in the join keys.

Developed by LogicBlox in the early 2010s.

LEAPFROG TRIEJOIN: A SIMPLE, WORST-CASE OPTIMAL JOIN ALGORITHM
ICDT 2014

LEAP-FROG TRIE

Relations must be indexes (or sorte
keys in advance before performing

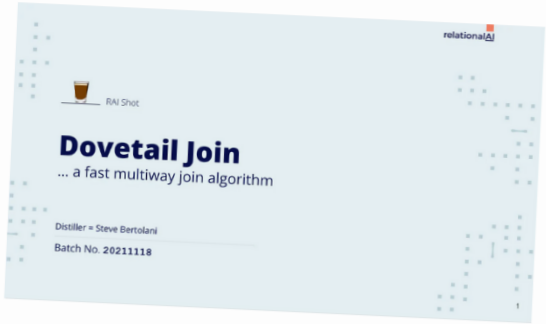
Represent relations with multiple a
→ One trie per relation.
→ Each level in the trie represents a singl
join keys.

Developed by LogicBlox in the ear

RelationalAI

Dovetail Join

November 22, 2021 by Steve Bertolani



At RelationalAI, we have built the first Relational Knowledge Graph System (RKGS).

One crucial part of our system is our dovetail join algorithm. This algorithm allows us to do a lot of complicated joins.... Extremely fast.

Why do you care about the Dovetail Join Algorithm? The dovetail algorithm is all about speed and it allows us to do more work, faster.

Dovetail Join requires fewer steps than traditional DBMS for complex queries. So it gets faster the harder or more complex the join is.

It also enables key technologies that allow us to speed up queries with high levels of optimization, memory reduction, and minimized code duplication.

Dovetail Join is a WCOJ (Worst Case Optimal Join) algorithm.... Which means we can mathematically prove that the more complicated the problem is, the faster we will go. Imagine working quality control on an assembly line. Instead of checking every single package or every other package, we skip most of the work and check every

LEAPFROG TRIEJOIN: A SIMPLE, WORST-CASE OPTIMAL JOIN ALGORITHM
ICDT 2014

LEAP-FROG TRIE JOIN

Table X

id
8
1
5
4
10
9
0
7
3
6

SORT!

Table Y

id
7
6
9
2
8
0

SORT!

Table Z

id
5
8
4
10
2

SORT!

LEAP-FROG TRIE JOIN

Table X

id
0
1
3
4
5
6
7
8
9
10

SORT!

Table Y

id
0
2
6
7
8
9

SORT!

Table Z

id
2
4
5
8
10

SORT!

LEAP-FROG TRIE JOIN

Table X

id
0
1
3
4
5
6
7
8
9
10

SORT!

Table Y

id
0
2
6
7
8
9

SORT!

Table Z

id
2
4
5
8
10

SORT!

$X \bowtie Y \bowtie Z$

X.id	0	1		3	4	5	6	7	8	9	10

X Iterator = 0

Y.id	0	2				6	7	8	9		

Y Iterator = 0

Z.id		2		4	5				8		10

Z Iterator = 2

LEAP-FROG TRIE JOIN

Table X

id
0
1
3
4
5
6
7
8
9
10

SORT!

Table Y

id
0
2
6
7
8
9

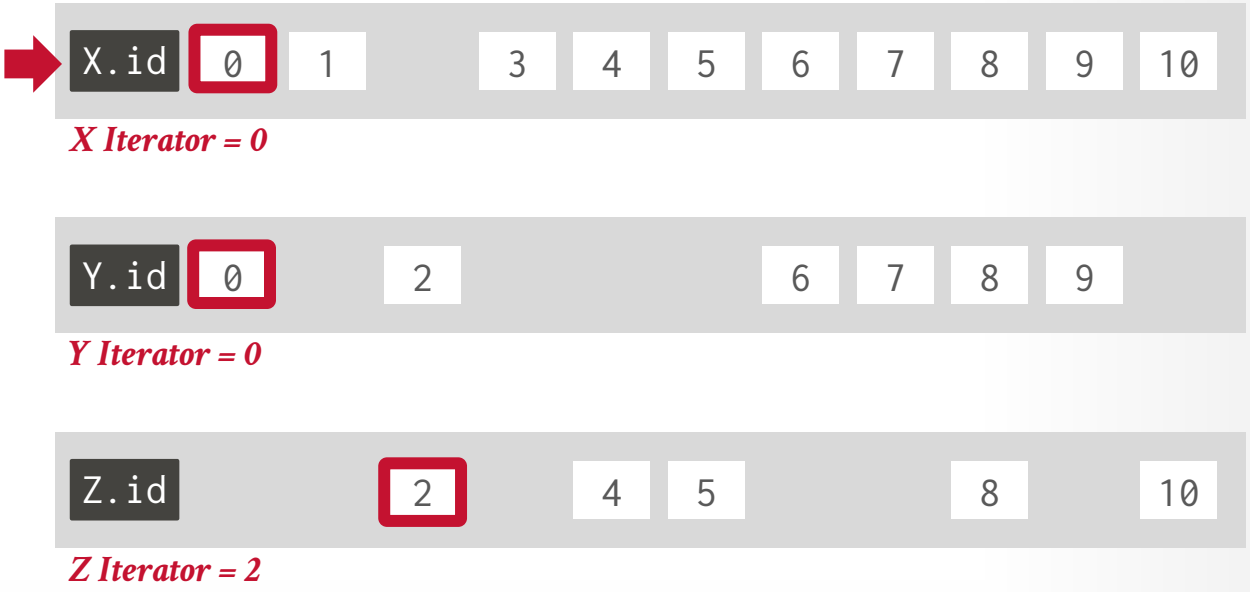
SORT!

Table Z

id
2
4
5
8
10

SORT!

$X \bowtie Y \bowtie Z$



LEAP-FROG TRIE JOIN

Table X

id
0
1
3
4
5
6
7
8
9
10

SORT!

Table Y

id
0
2
6
7
8
9

SORT!

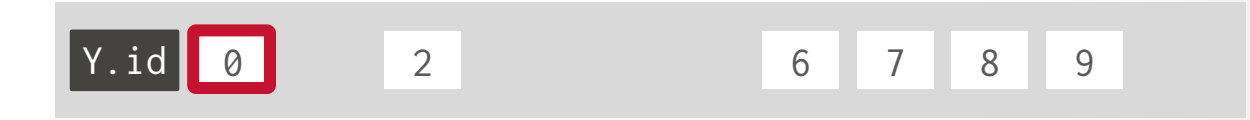
Table Z

id
2
4
5
8
10

SORT!

$X \bowtie Y \bowtie Z$

X Iterator = 3



Y Iterator = 0

Z Iterator = 2

LEAP-FROG TRIE JOIN

Table X

id
0
1
3
4
5
6
7
8
9
10

SORT!

Table Y

id
0
2
6
7
8
9

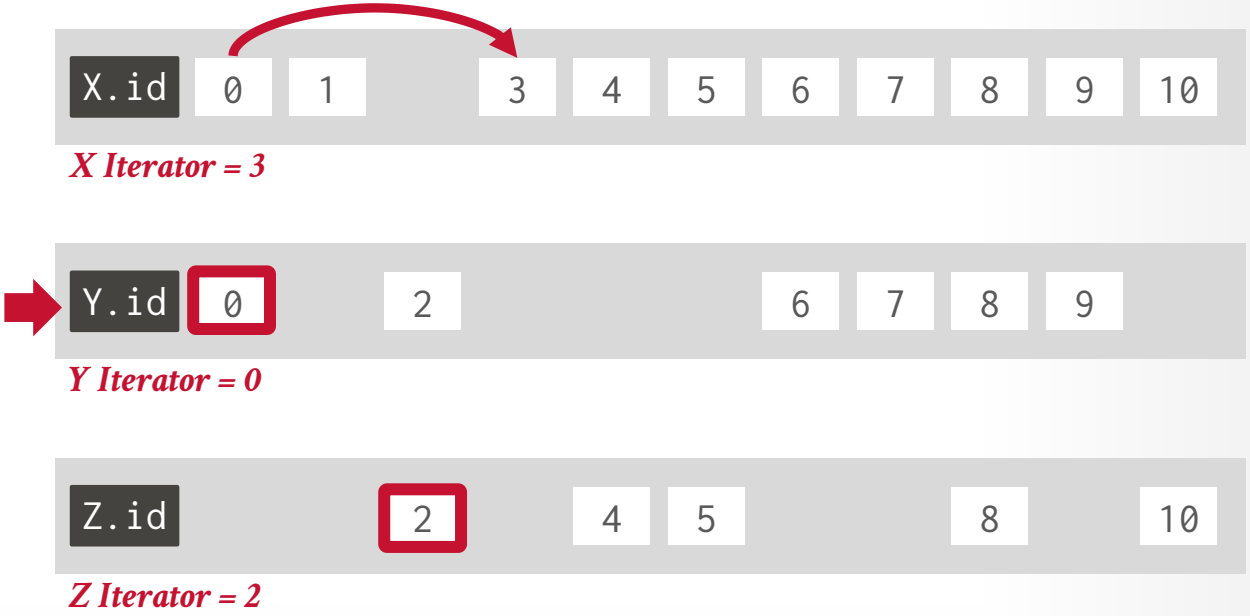
SORT!

Table Z

id
2
4
5
8
10

SORT!

$X \bowtie Y \bowtie Z$



LEAP-FROG TRIE JOIN

Table X

id
0
1
3
4
5
6
7
8
9
10

SORT!

Table Y

id
0
2
6
7
8
9

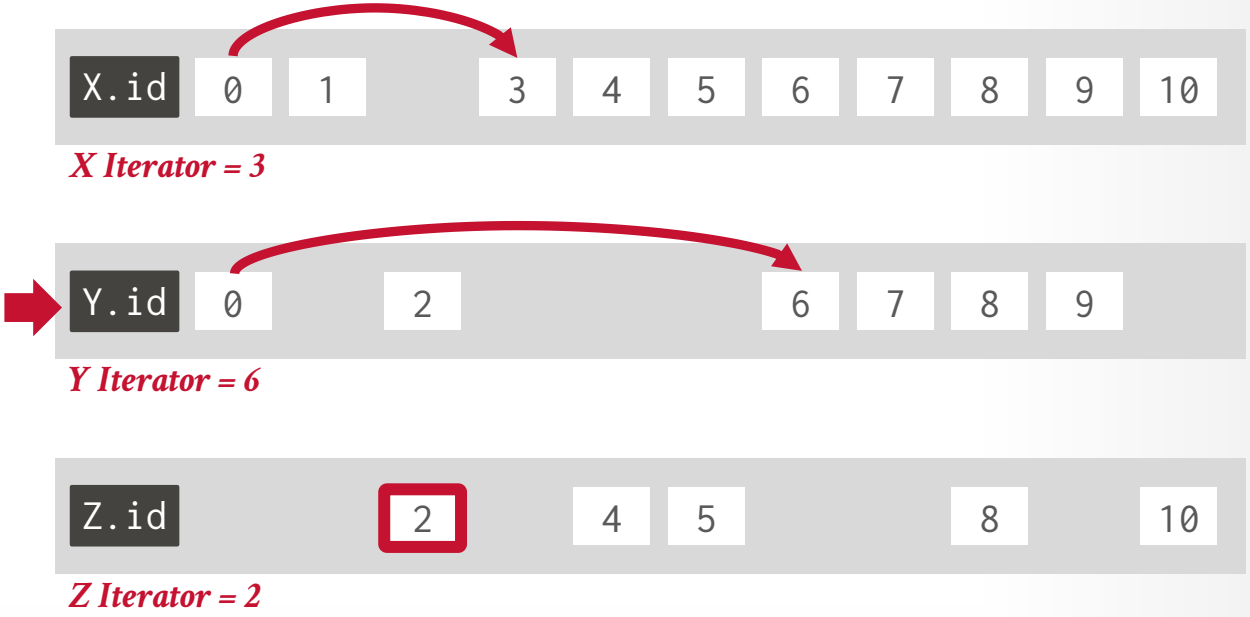
SORT!

Table Z

id
2
4
5
8
10

SORT!

$X \bowtie Y \bowtie Z$



LEAP-FROG TRIE JOIN

Table X

id
0
1
3
4
5
6
7
8
9
10

SORT!

Table Y

id
0
2
6
7
8
9

SORT!

Table Z

id
2
4
5
8
10

SORT!

$X \bowtie Y \bowtie Z$

X Iterator = 3

Y Iterator = 6

Z Iterator = 2

LEAP-FROG TRIE JOIN

Table X

id
0
1
3
4
5
6
7
8
9
10

SORT!

Table Y

id
0
2
6
7
8
9

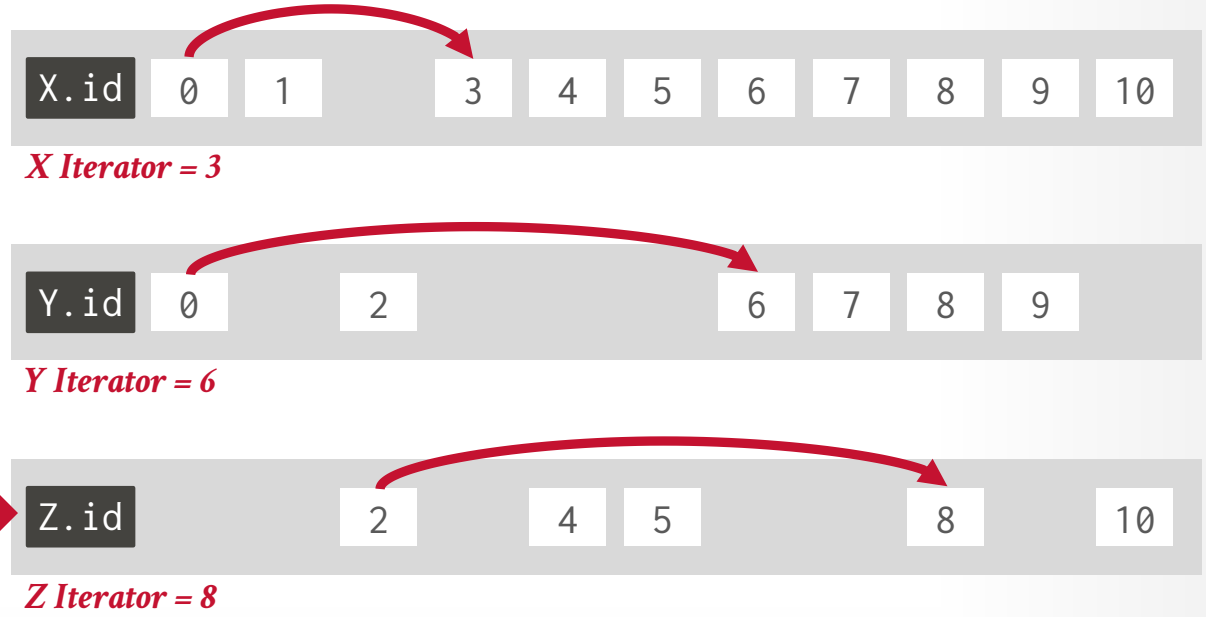
SORT!

Table Z

id
2
4
5
8
10

SORT!

$X \bowtie Y \bowtie Z$



LEAP-FROG TRIE JOIN

Table X

id
0
1
3
4
5
6
7
8
9
10

SORT!

Table Y

id
0
2
6
7
8
9

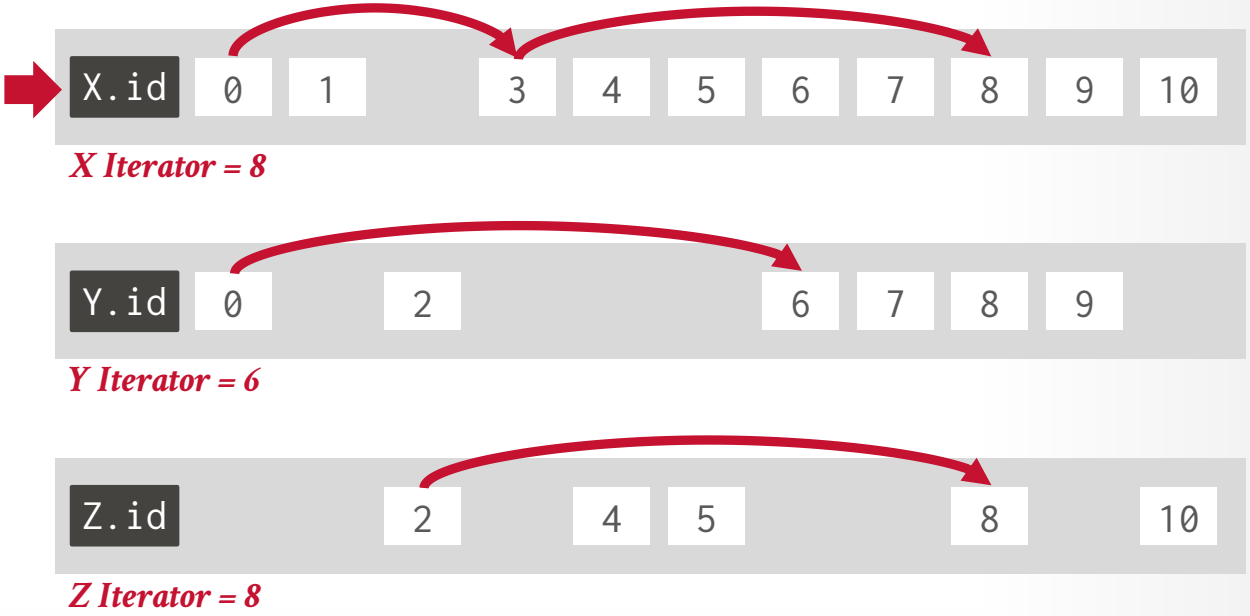
SORT!

Table Z

id
2
4
5
8
10

SORT!

$X \bowtie Y \bowtie Z$



LEAP-FROG TRIE JOIN

Table X

id
0
1
3
4
5
6
7
8
9
10

SORT!

Table Y

id
0
2
6
7
8
9

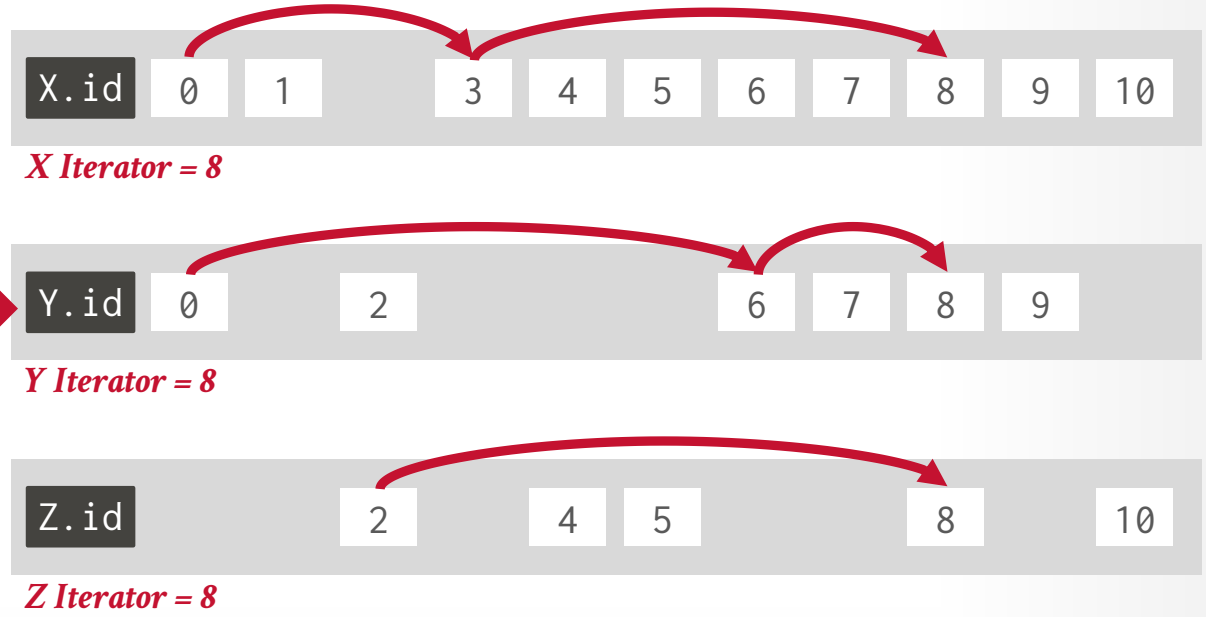
SORT!

Table Z

id
2
4
5
8
10

SORT!

$X \bowtie Y \bowtie Z$



LEAP-FROG TRIE JOIN

Table X

id
0
1
3
4
5
6
7
8
9
10

SORT!

Table Y

id
0
2
6
7
8
9

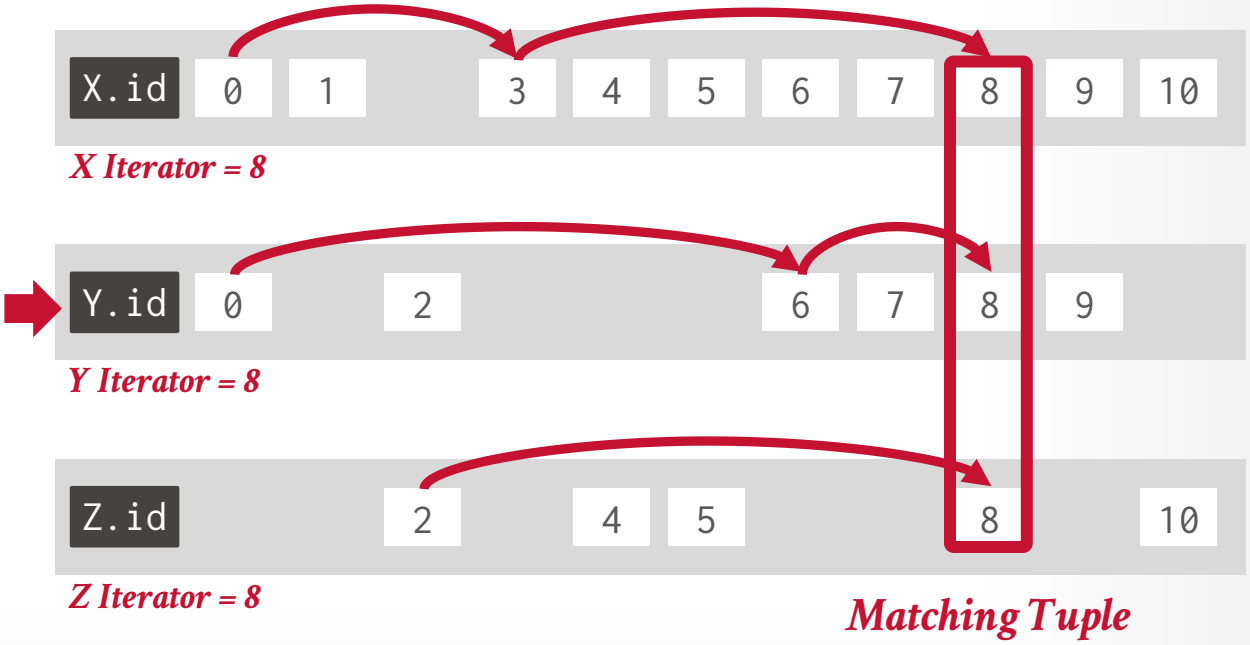
SORT!

Table Z

id
2
4
5
8
10

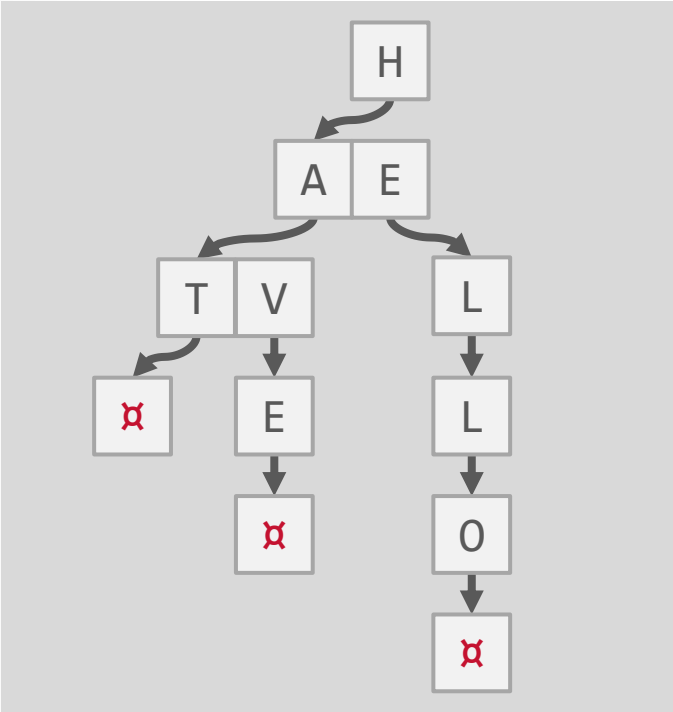
SORT!

$X \bowtie Y \bowtie Z$



TRIE

Keys: HELLO, HAT, HAVE



Use a digital representation of keys to examine prefixes one-by-one instead of comparing entire key.

→ Also known as ***Digital Search Tree***, ***Prefix Tree***.

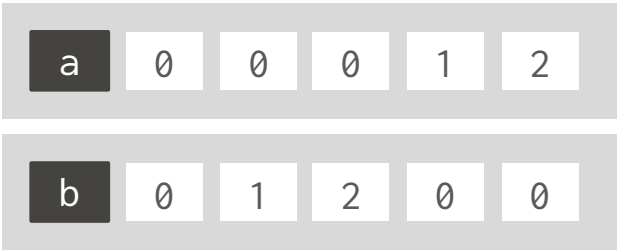
Contrast this with a B+Tree that stores all the digits of a key together in a node.

LEAP-FROG TRIE JOIN

Represent target relations with multiple join key attributes as tries where each attribute is a different level in the data structure.

Table R

a	b
0	0
0	1
0	2
1	0
2	0

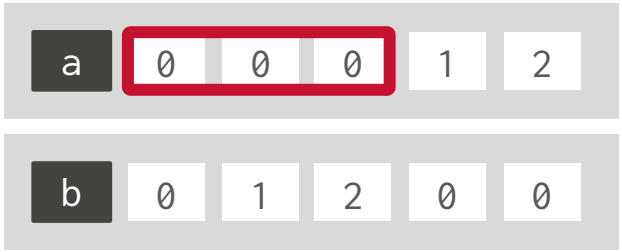


LEAP-FROG TRIE JOIN

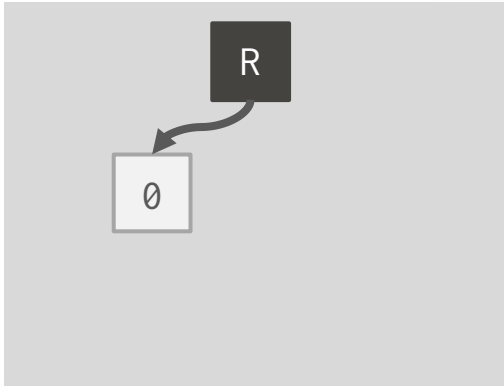
Represent target relations with multiple join key attributes as tries where each attribute is a different level in the data structure.

Table R

a	b
0	0
0	1
0	2
1	0
2	0



Trie

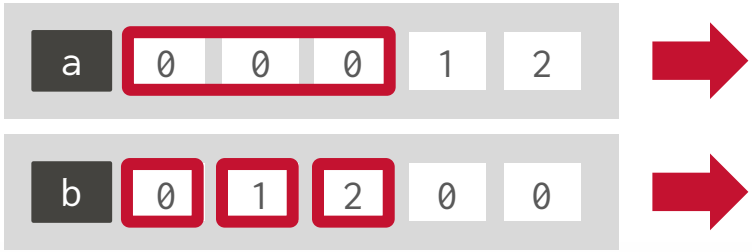


LEAP-FROG TRIE JOIN

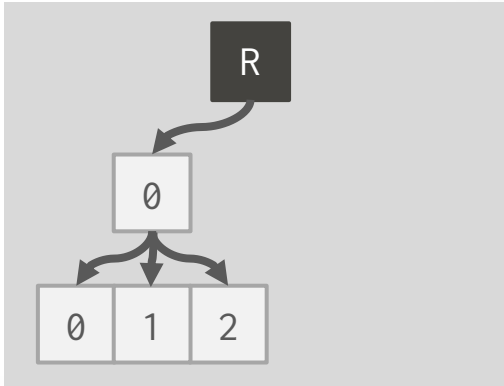
Represent target relations with multiple join key attributes as tries where each attribute is a different level in the data structure.

Table R

a	b
0	0
0	1
0	2
1	0
2	0



Trie

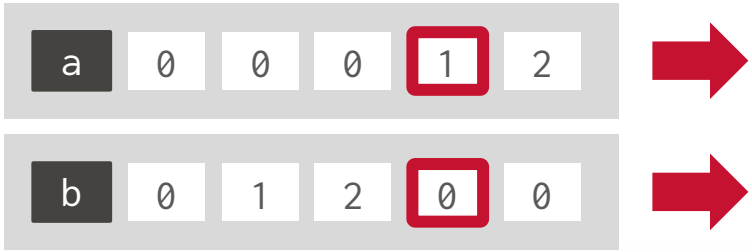


LEAP-FROG TRIE JOIN

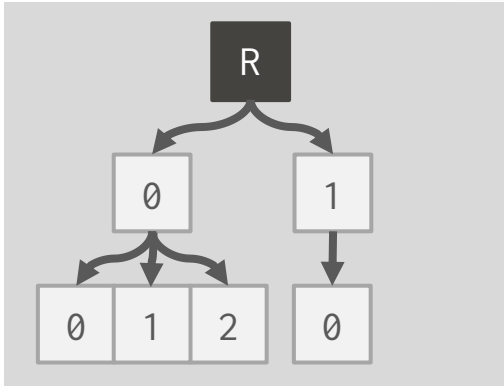
Represent target relations with multiple join key attributes as tries where each attribute is a different level in the data structure.

Table R

a	b
0	0
0	1
0	2
1	0
2	0



Trie

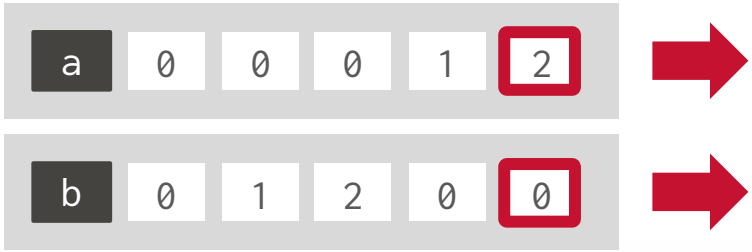


LEAP-FROG TRIE JOIN

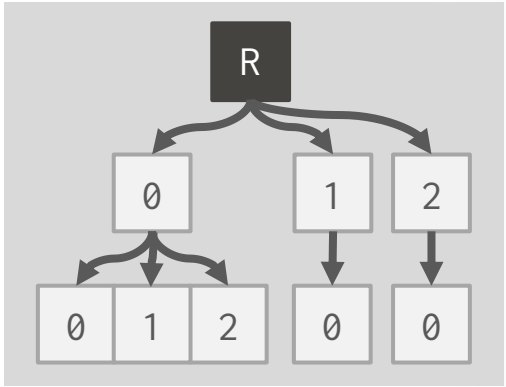
Represent target relations with multiple join key attributes as tries where each attribute is a different level in the data structure.

Table R

a	b
0	0
0	1
0	2
1	0
2	0



Trie

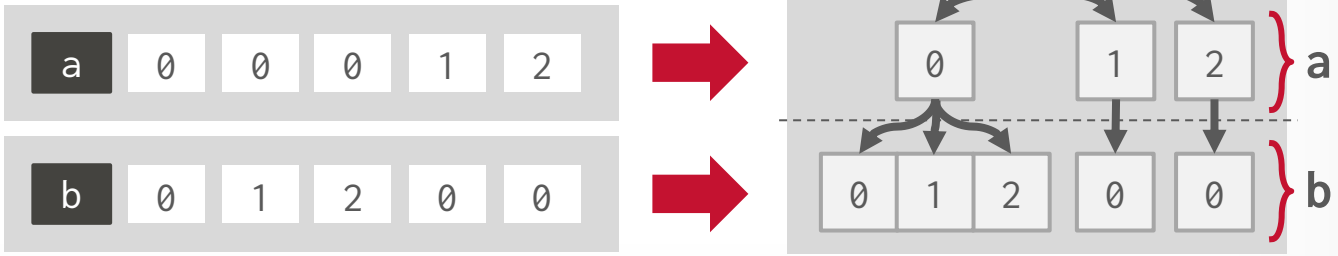


LEAP-FROG TRIE JOIN

Represent target relations with multiple join key attributes as tries where each attribute is a different level in the data structure.

Table R

a	b
0	0
0	1
0	2
1	0
2	0



LEAP-FROG TRIE JOIN

Table R

a	b
0	0
0	1
0	2
1	0
2	0

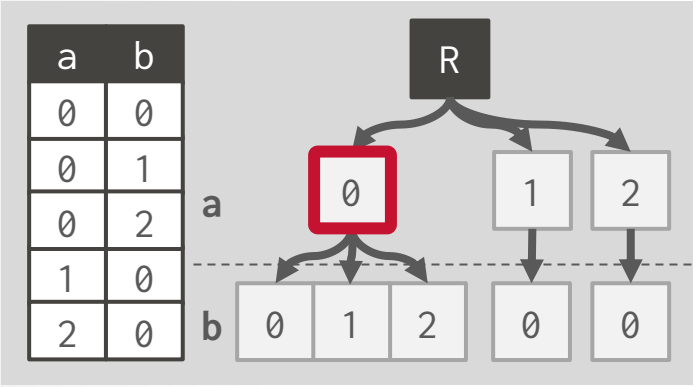


Table S

b	c
0	0
0	1
0	2
1	0
2	0

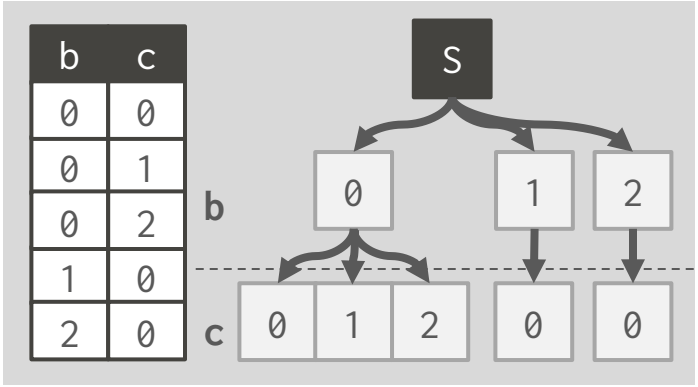
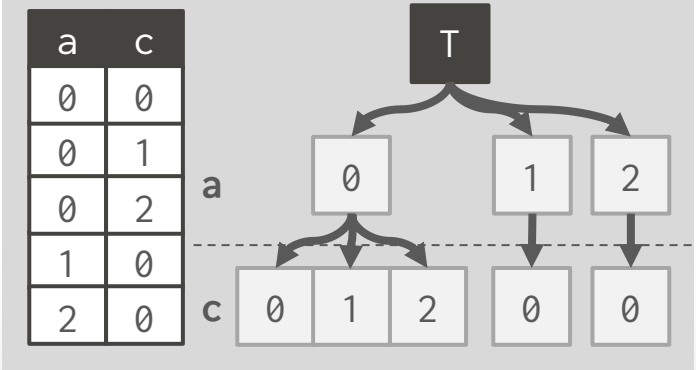


Table T

a	c
0	0
0	1
0	2
1	0
2	0




```
SELECT *
  FROM R, S, T
 WHERE R.a = T.a
   AND R.b = S.b
   AND S.c = T.c
```

$R \bowtie S \bowtie T$

a	b	c

Source: Richard Gankema

LEAP-FROG TRIE JOIN

Table R

a	b
0	0
0	1
0	2
1	0
2	0

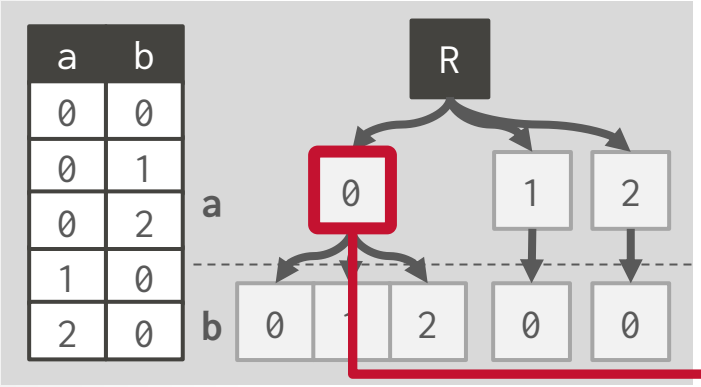


Table S

b	c
0	0
0	1
0	2
1	0
2	0

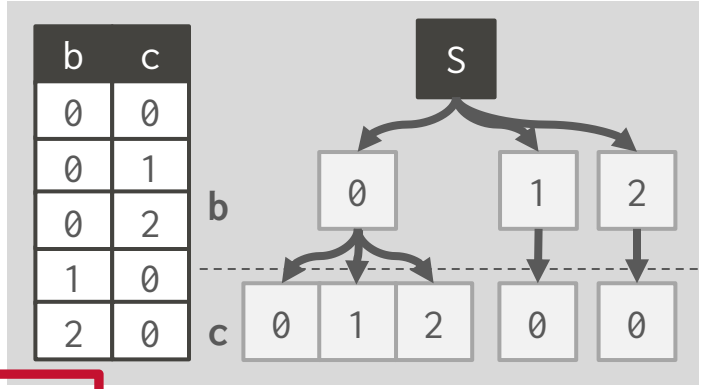
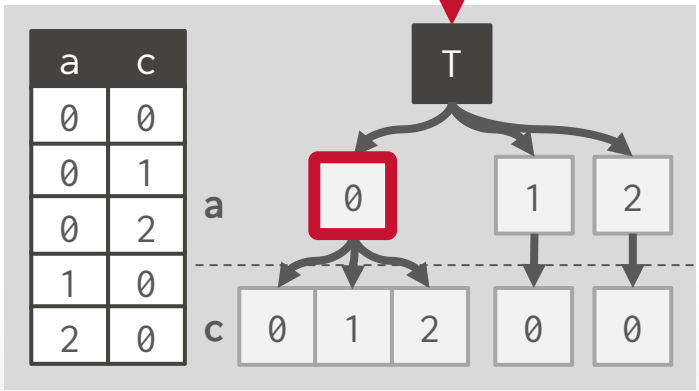


Table T

a	c
0	0
0	1
0	2
1	0
2	0




```
SELECT *
  FROM R, S, T
 WHERE R.a = T.a
   AND R.b = S.b
   AND S.c = T.c
```

$R \bowtie S \bowtie T$

a	b	c

Source: Richard Gankema

LEAP-FROG TRIE JOIN

Table R

a	b
0	0
0	1
0	2
1	0
2	0

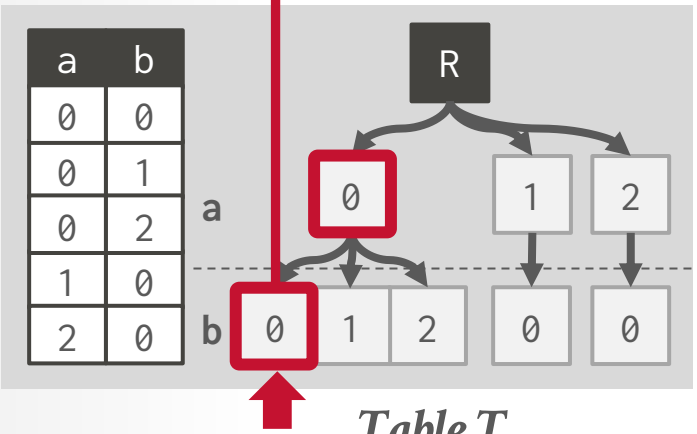


Table S

b	c
0	0
0	1
0	2
1	0
2	0

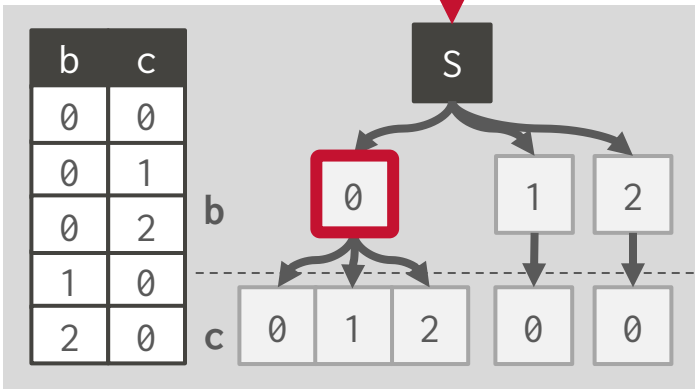
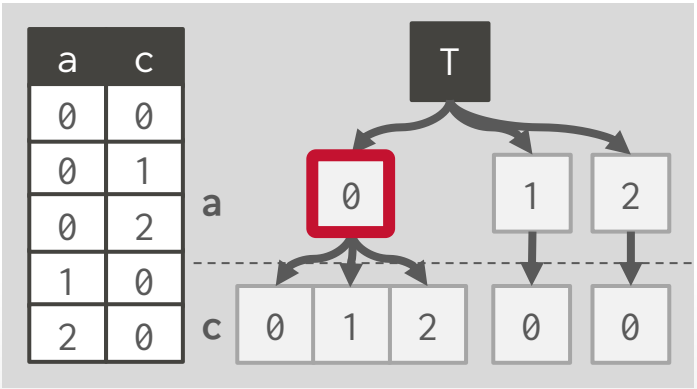


Table T

a	c
0	0
0	1
0	2
1	0
2	0




```
SELECT *
  FROM R, S, T
 WHERE R.a = T.a
   AND R.b = S.b
   AND S.c = T.c
```

$R \bowtie S \bowtie T$

a	b	c

Source: Richard Gankema

LEAP-FROG TRIE JOIN

Table R

a	b
0	0
0	1
0	2
1	0
2	0

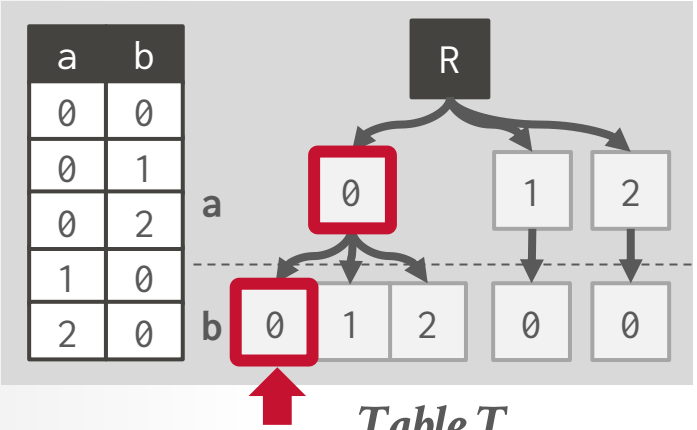


Table S

b	c
0	0
0	1
0	2
1	0
2	0

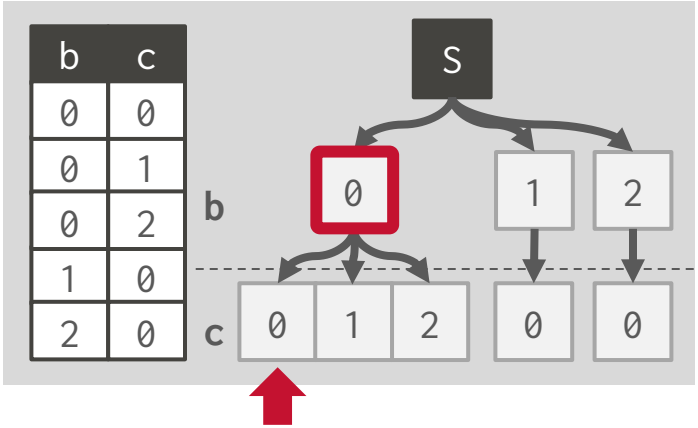
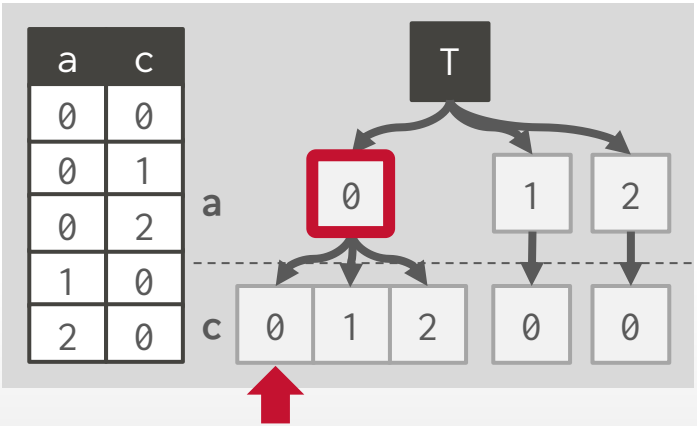


Table T

a	c
0	0
0	1
0	2
1	0
2	0




```
SELECT *
  FROM R, S, T
 WHERE R.a = T.a
   AND R.b = S.b
   AND S.c = T.c
```

$R \bowtie S \bowtie T$

a	b	c

Source: Richard Gankema

LEAP-FROG TRIE JOIN

Table R

a	b
0	0
0	1
0	2
1	0
2	0

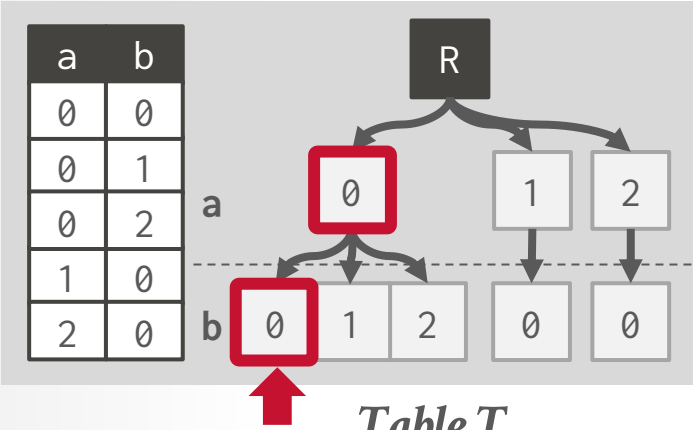


Table S

b	c
0	0
0	1
0	2
1	0
2	0

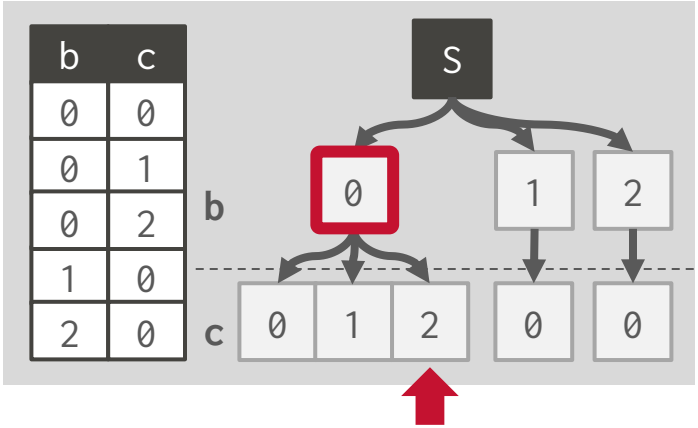
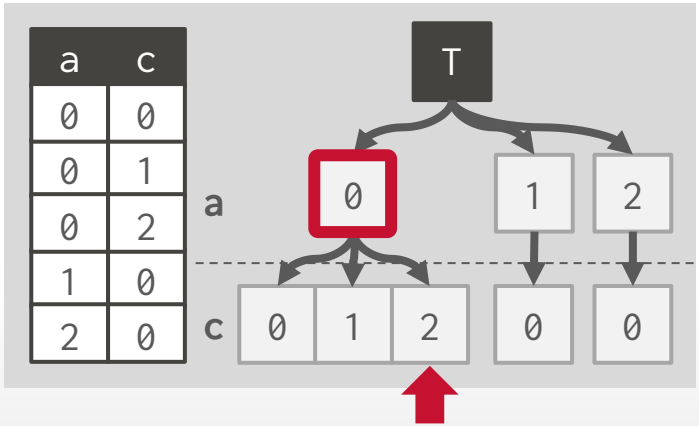


Table T

a	c
0	0
0	1
0	2
1	0
2	0



$S.c \rightarrow (0,1,2)$

$\cap$

$T.c \rightarrow (0,1,2)$

```
SELECT *
  FROM R, S, T
 WHERE R.a = T.a
   AND R.b = S.b
   AND S.c = T.c
```

$R \bowtie S \bowtie T$

a	b	c
0	0	0
0	0	1
0	0	2

Source: Richard Gankema

LEAP-FROG TRIE JOIN

Table R

a	b
0	0
0	1
0	2
1	0
2	0

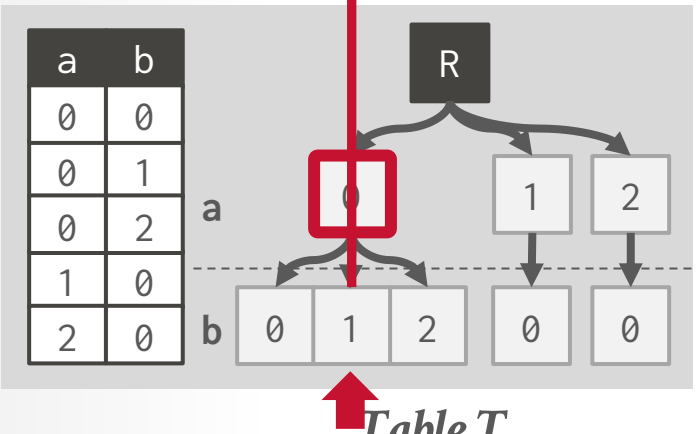


Table S

b	c
0	0
0	1
0	2
1	0
2	0

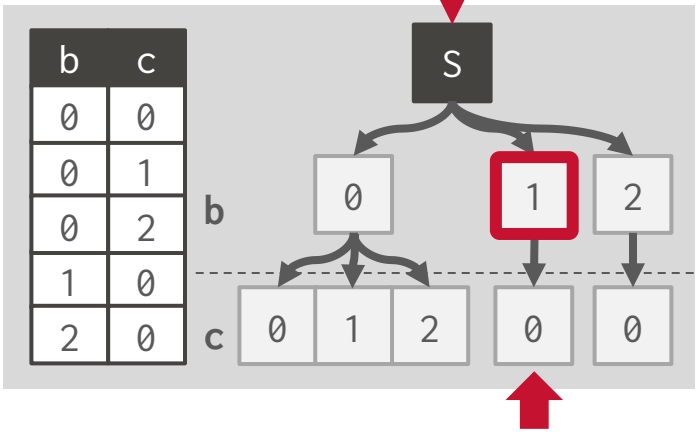
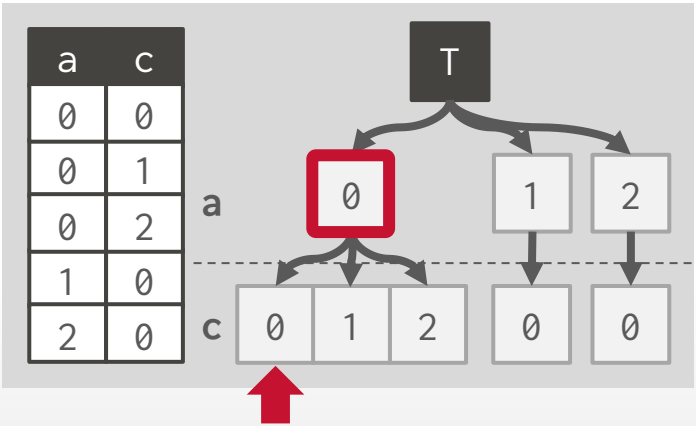


Table T

a	c
0	0
0	1
0	2
1	0
2	0




```
SELECT *
  FROM R, S, T
 WHERE R.a = T.a
   AND R.b = S.b
   AND S.c = T.c
```

$R \bowtie S \bowtie T$

a	b	c
0	0	0
0	0	1
0	0	2

Source: Richard Gankema

LEAP-FROG TRIE JOIN

Table R

a	b
0	0
0	1
0	2
1	0
2	0

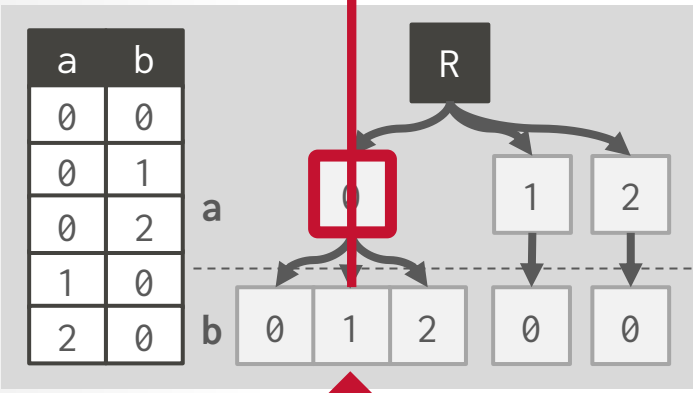


Table S

b	c
0	0
0	1
0	2
1	0
2	0

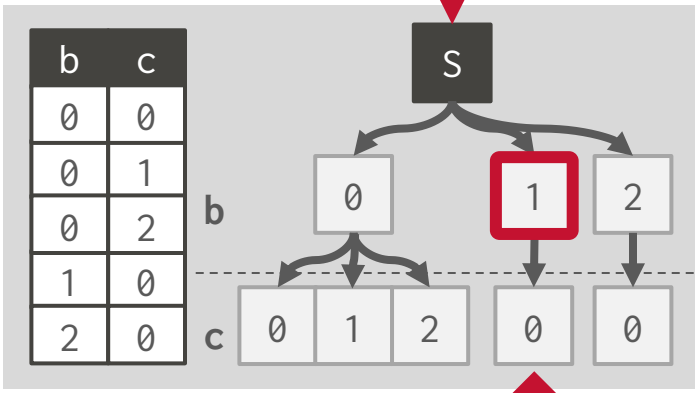
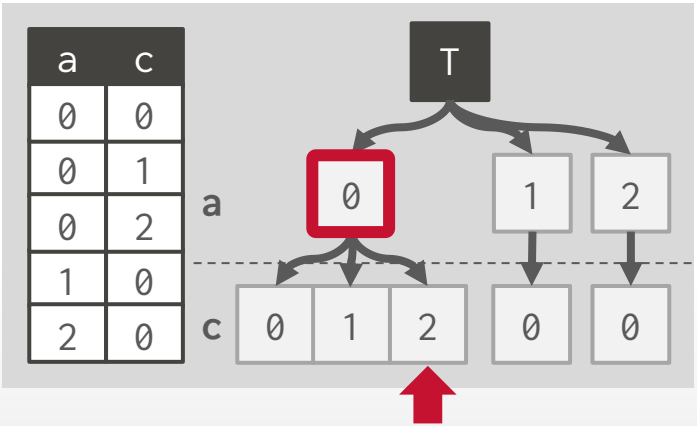


Table T

a	c
0	0
0	1
0	2
1	0
2	0



$S.c \rightarrow (0)$

$\cap$

$T.c \rightarrow (0, 1, 2)$

```
SELECT *
  FROM R, S, T
 WHERE R.a = T.a
   AND R.b = S.b
   AND S.c = T.c
```

$R \bowtie S \bowtie T$

a	b	c
0	0	0
0	0	1
0	0	2
0	1	0

Source: Richard Gankema

LEAP-FROG TRIE JOIN

Table R

a	b
0	0
0	1
0	2
1	0
2	0

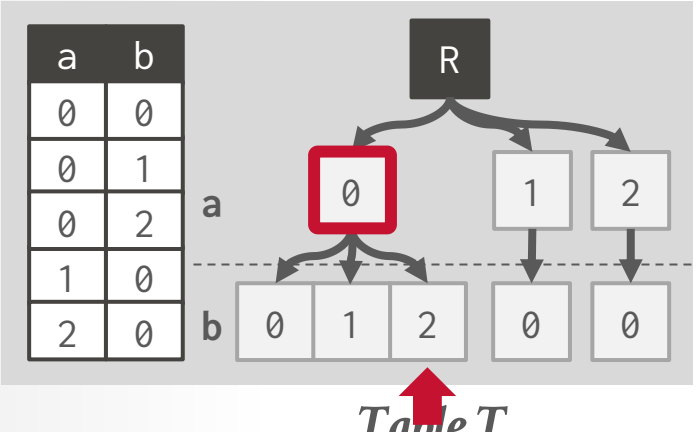


Table S

b	c
0	0
0	1
0	2
1	0
2	0

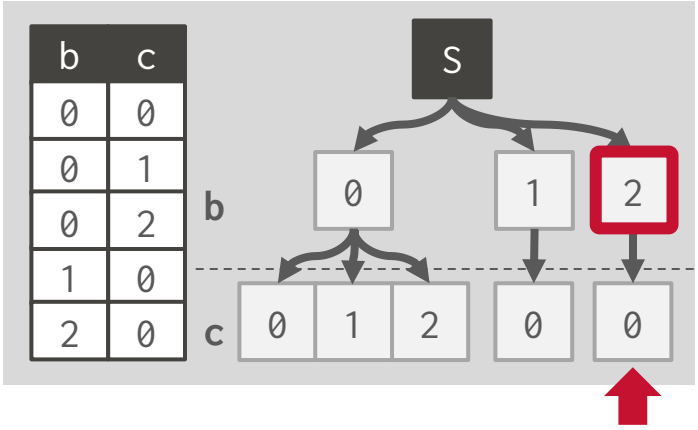
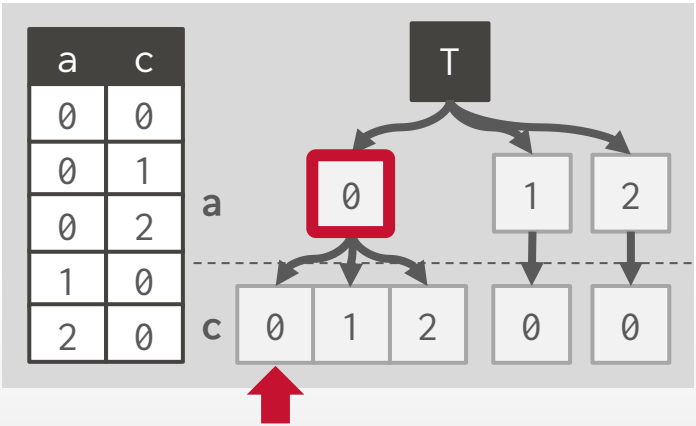


Table T

a	c
0	0
0	1
0	2
1	0
2	0




```
SELECT *
  FROM R, S, T
 WHERE R.a = T.a
   AND R.b = S.b
   AND S.c = T.c
```

$R \bowtie S \bowtie T$

a	b	c
0	0	0
0	0	1
0	0	2
0	1	0

Source: Richard Gankema

LEAP-FROG TRIE JOIN

Table R

a	b
0	0
0	1
0	2
1	0
2	0

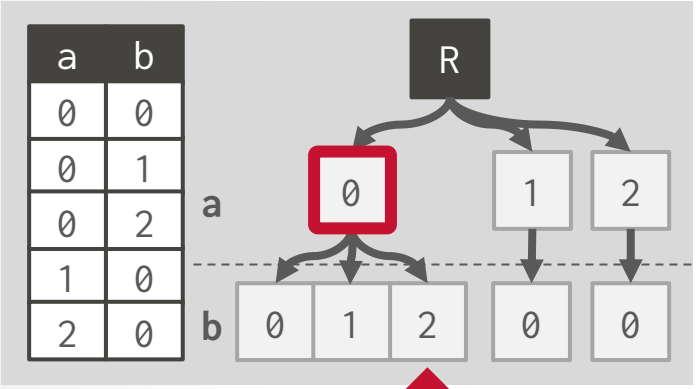


Table S

b	c
0	0
0	1
0	2
1	0
2	0

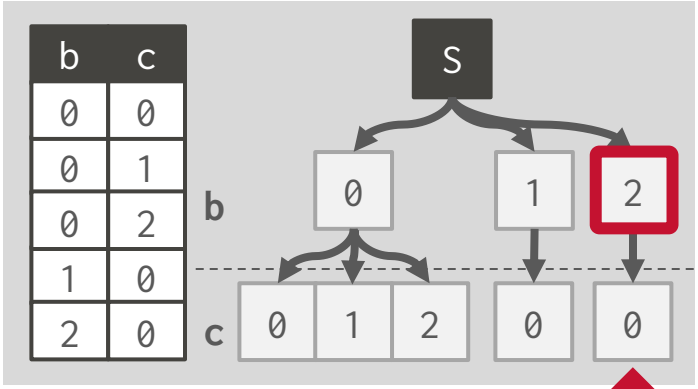
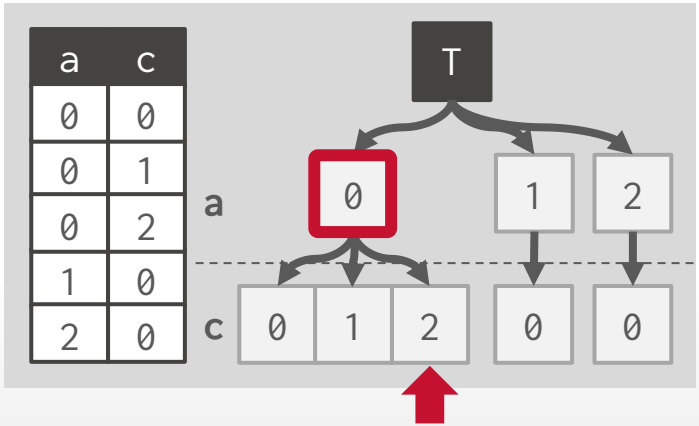


Table T

a	c
0	0
0	1
0	2
1	0
2	0



$S.c \rightarrow (0)$

$\cap$

$T.c \rightarrow (0, 1, 2)$

```
SELECT *
  FROM R, S, T
 WHERE R.a = T.a
   AND R.b = S.b
   AND S.c = T.c
```

$R \bowtie S \bowtie T$

a	b	c
0	0	0
0	0	1
0	0	2
0	1	0
0	2	0

Source: Richard Gankema

LEAP-FROG TRIE JOIN

Table R

a	b
0	0
0	1
0	2
1	0
2	0

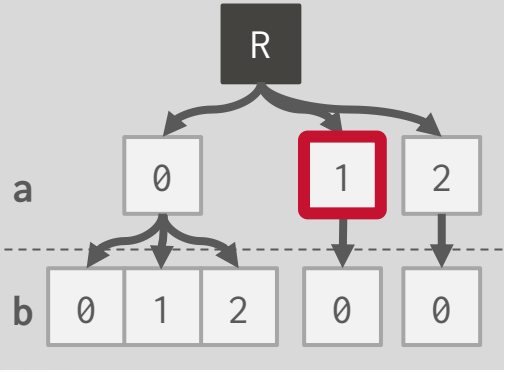


Table S

b	c
0	0
0	1
0	2
1	0
2	0

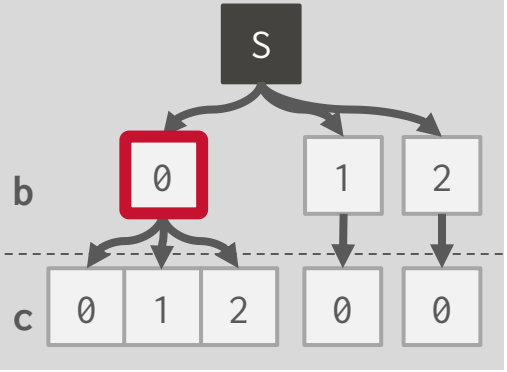
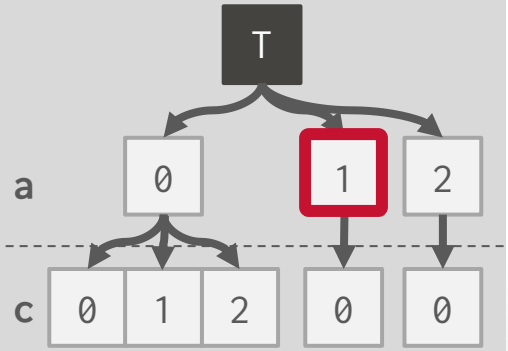


Table T

a	c
0	0
0	1
0	2
1	0
2	0




```
SELECT *
  FROM R, S, T
 WHERE R.a = T.a
   AND R.b = S.b
   AND S.c = T.c
```

$R \bowtie S \bowtie T$

a	b	c
0	0	0
0	0	1
0	0	2
0	1	0
0	2	0
1	0	0

Source: Richard Gankema

LEAP-FROG TRIE JOIN

Table R

a	b
0	0
0	1
0	2
1	0
2	0

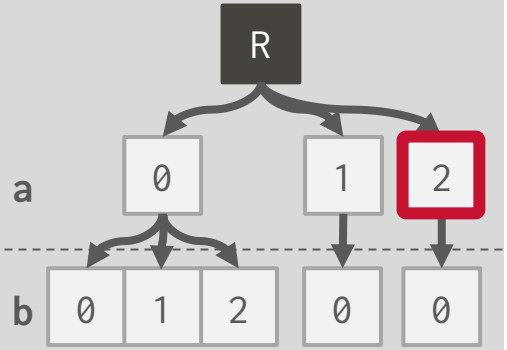
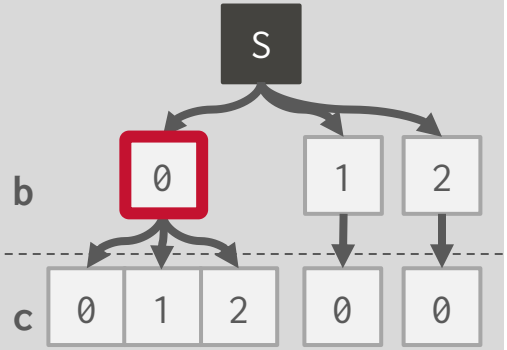


Table S

b	c
0	0
0	1
0	2
1	0
2	0



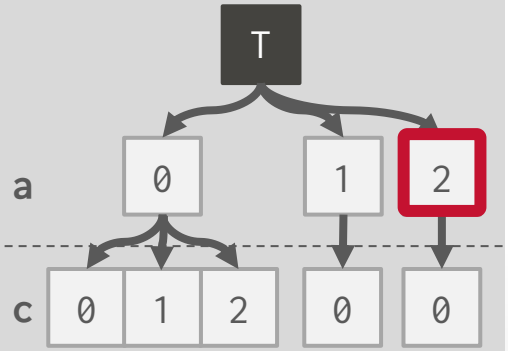

```
SELECT *
  FROM R, S, T
 WHERE R.a = T.a
   AND R.b = S.b
   AND S.c = T.c
```

$R \bowtie S \bowtie T$

a	b	c
0	0	0
0	0	1
0	0	2
0	1	0
0	2	0
1	0	0
2	0	0

Table T

a	c
0	0
0	1
0	2
1	0
2	0



Source: Richard Gankema

OBSERVATION

Building a trie for every relation on the fly is expensive.

→ Even if the database is read-only, building tries for every possible join ordering is impractical.

An alternative approach is to use nested hash tables, but this is also expensive:

→ At least one key comparison to detect hash collisions.

→ Need to store the actual keys or pointers to the tuples to deal with collisions. Lots of cache misses.

→ Need to deal with variable-length keys using dictionary encoding.

MULTI-WAY HASH TRIE JOINS

Instead of storing join key attribute values in the look-up data structure, the DBMS stores hash values in a trie.

→ Each trie node is a hash table that maps hash values to either child nodes or tuples.

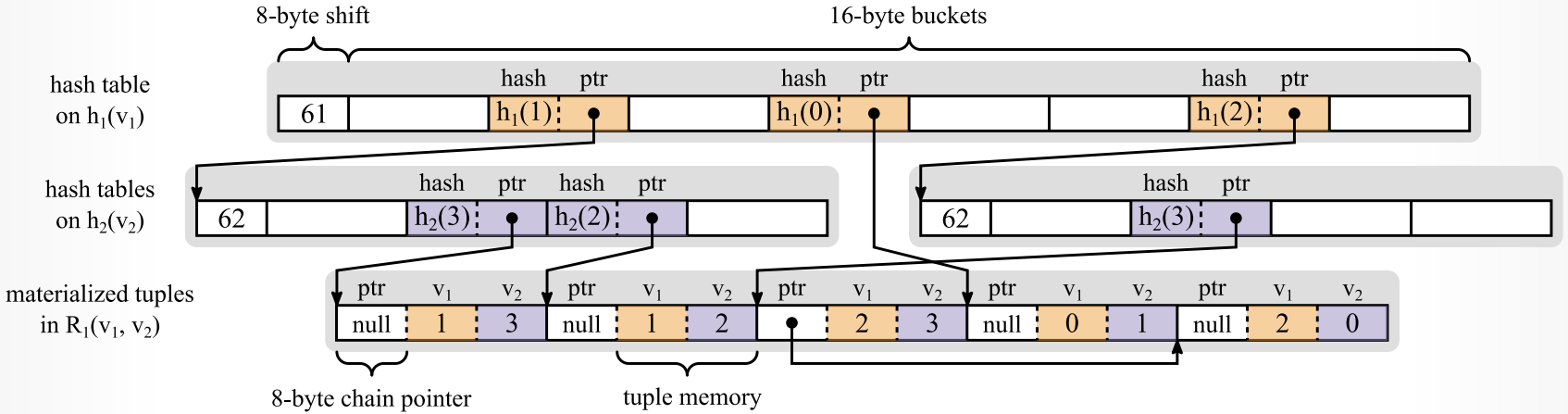
No type-specific logic for computing set intersections and lookup operations.

→ DBMS only uses fast integer comparisons.

→ Need to remove false positive matches on hash collisions.

ADOPTING WORST-CASE OPTIMAL JOINS IN RELATIONAL DATABASE SYSTEMS
SIGMOD 2020

HASH TRIE



Source: Michael Freitag

HASH TRIE: TAGGED POINTERS

Exploit unused portion of 64-bit pointers to trie nodes to store additional meta-data.

→ x86-64 only uses 48 bits for memory addresses.

HASH TRIE: TAGGED POINTERS

Exploit unused portion of 64-bit pointers to trie nodes to store additional meta-data.

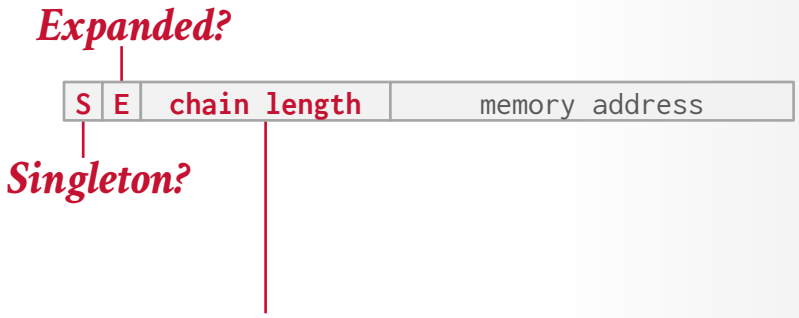
→ x86-64 only uses 48 bits for memory addresses.

1-bit: Singleton Flag

1-bit: Expansion Flag

14-bits: Chain Length

48-bits: Memory Address

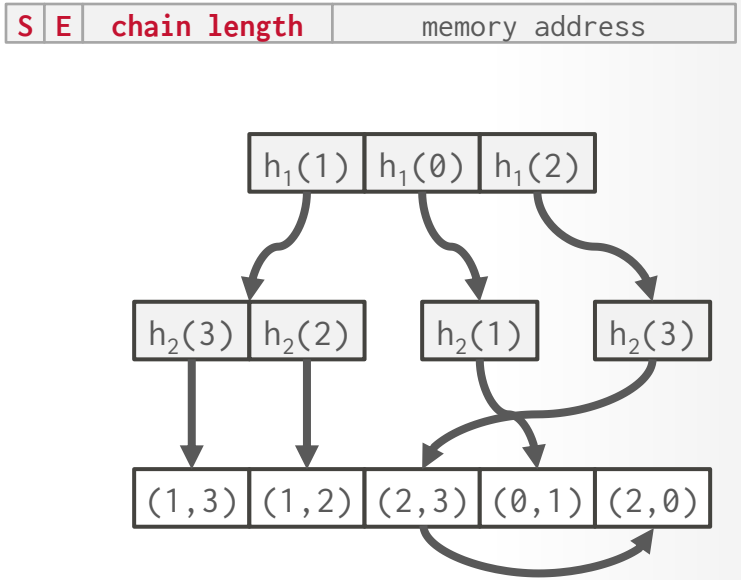


HASH TRIE: SINGLETON PRUNING

The size of hash tables in the trie get smaller in the lower trie levels.

→ Entries in inner nodes often point to only a single tuple below it.

Optimization: Instead of storing each level of trie for these paths, only store a pointer directly to the singleton tuple.

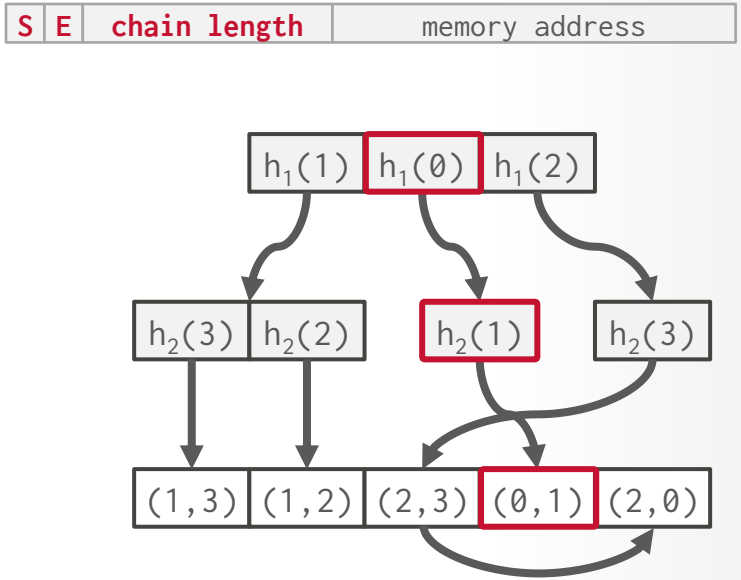


HASH TRIE: SINGLETON PRUNING

The size of hash tables in the trie get smaller in the lower trie levels.

→ Entries in inner nodes often point to only a single tuple below it.

Optimization: Instead of storing each level of trie for these paths, only store a pointer directly to the singleton tuple.

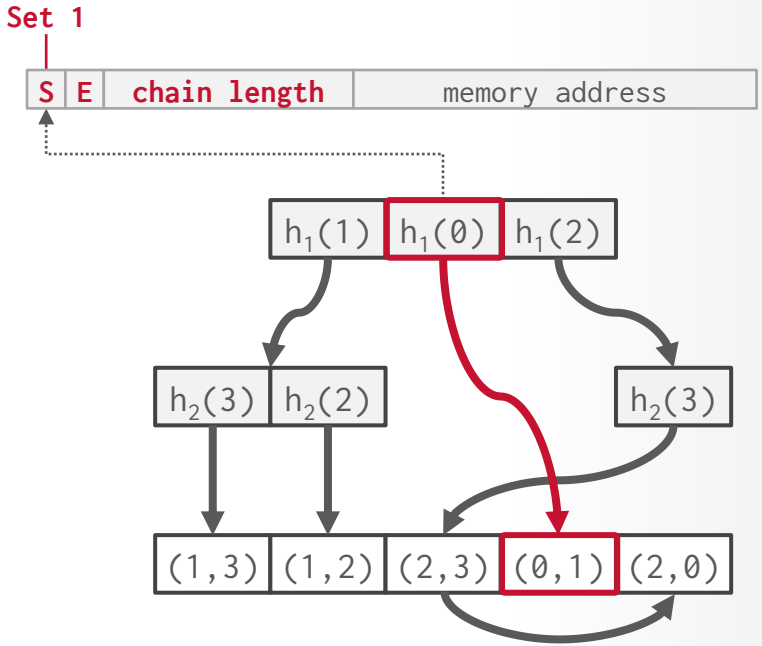


HASH TRIE: SINGLETON PRUNING

The size of hash tables in the trie get smaller in the lower trie levels.

→ Entries in inner nodes often point to only a single tuple below it.

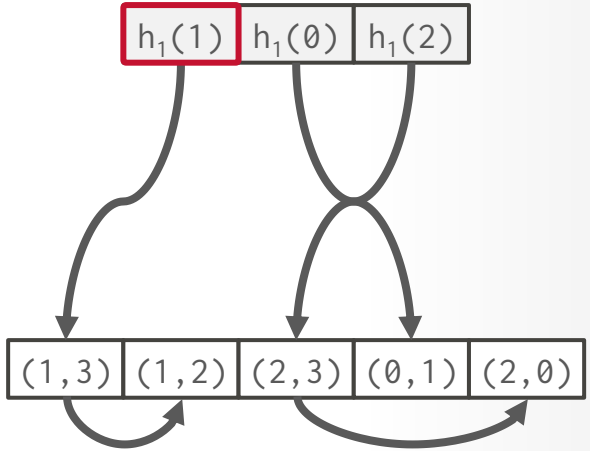
Optimization: Instead of storing each level of trie for these paths, only store a pointer directly to the singleton tuple.



HASH TRIE: LAZY CHILD EXPANSION

Depending on the selectivity of the intersection operation, the DBMS never accesses many inner nodes.

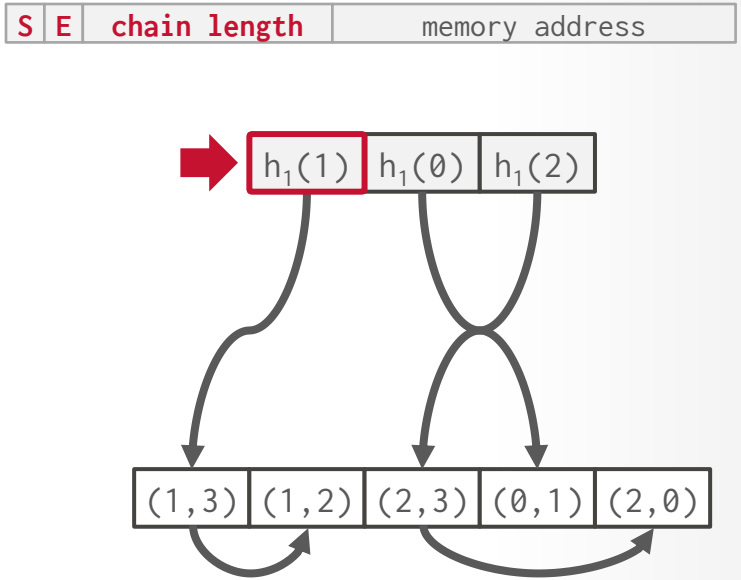
Optimization: Only materialize inner nodes in the trie when they are accessed during the probe phase.
→ Must always create root node.



HASH TRIE: LAZY CHILD EXPANSION

Depending on the selectivity of the intersection operation, the DBMS never accesses many inner nodes.

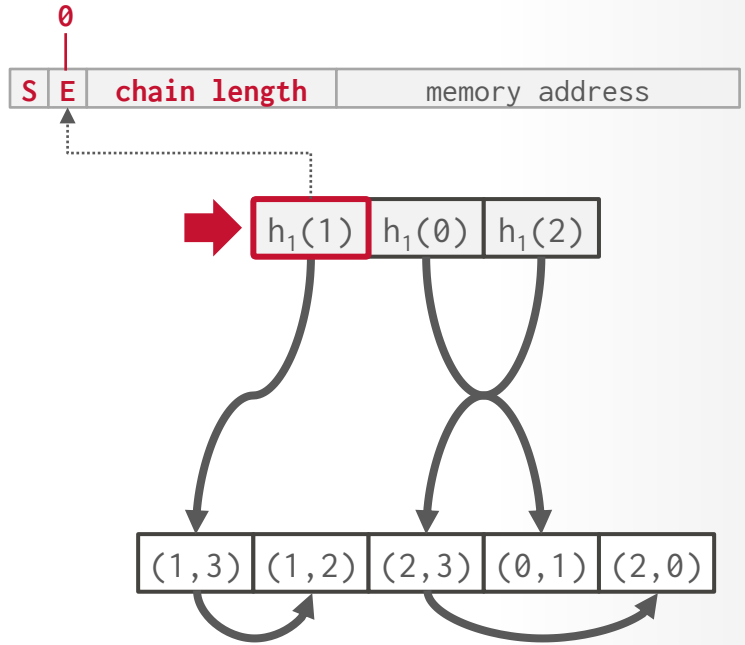
Optimization: Only materialize inner nodes in the trie when they are accessed during the probe phase.
→ Must always create root node.



HASH TRIE: LAZY CHILD EXPANSION

Depending on the selectivity of the intersection operation, the DBMS never accesses many inner nodes.

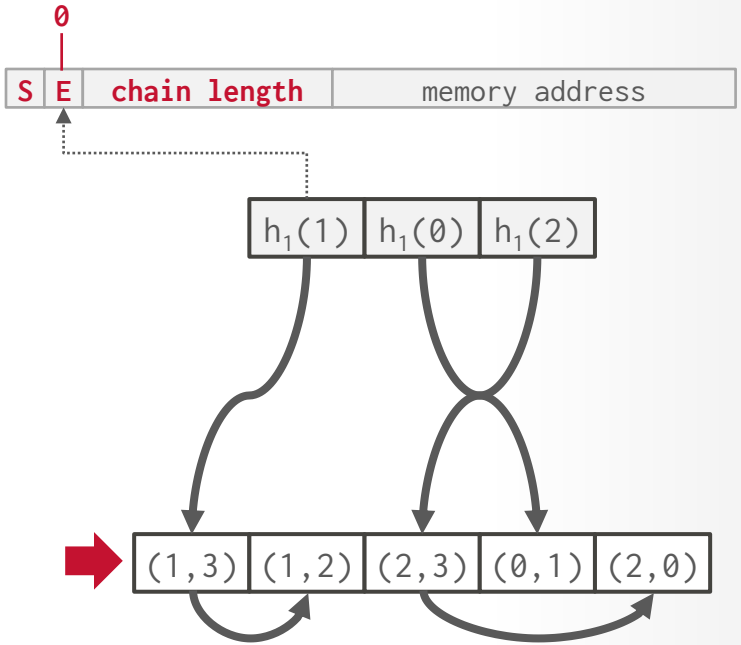
Optimization: Only materialize inner nodes in the trie when they are accessed during the probe phase.
→ Must always create root node.



HASH TRIE: LAZY CHILD EXPANSION

Depending on the selectivity of the intersection operation, the DBMS never accesses many inner nodes.

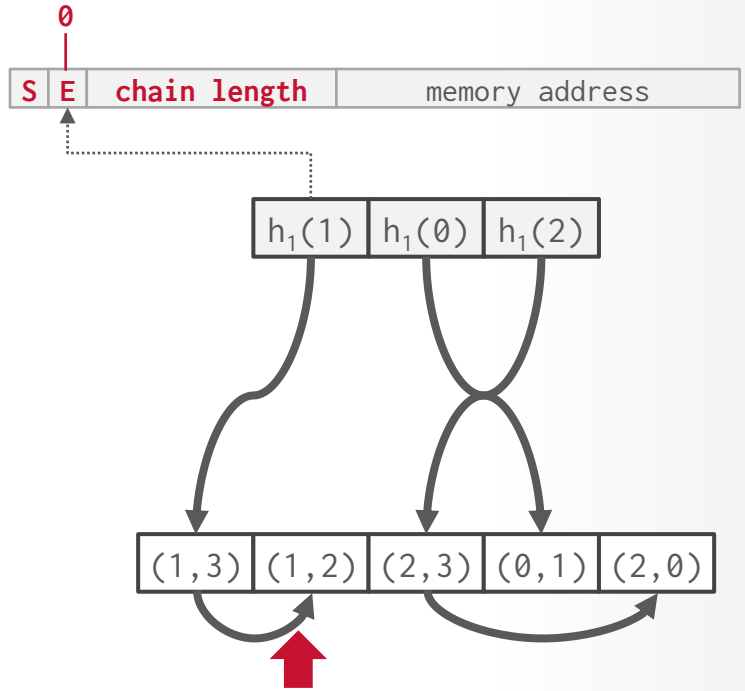
Optimization: Only materialize inner nodes in the trie when they are accessed during the probe phase.
→ Must always create root node.



HASH TRIE: LAZY CHILD EXPANSION

Depending on the selectivity of the intersection operation, the DBMS never accesses many inner nodes.

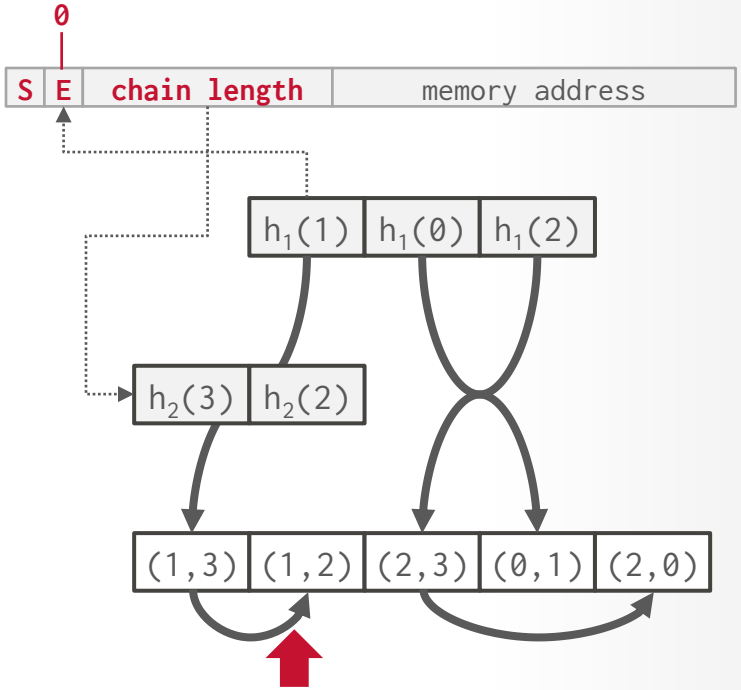
Optimization: Only materialize inner nodes in the trie when they are accessed during the probe phase.
→ Must always create root node.



HASH TRIE: LAZY CHILD EXPANSION

Depending on the selectivity of the intersection operation, the DBMS never accesses many inner nodes.

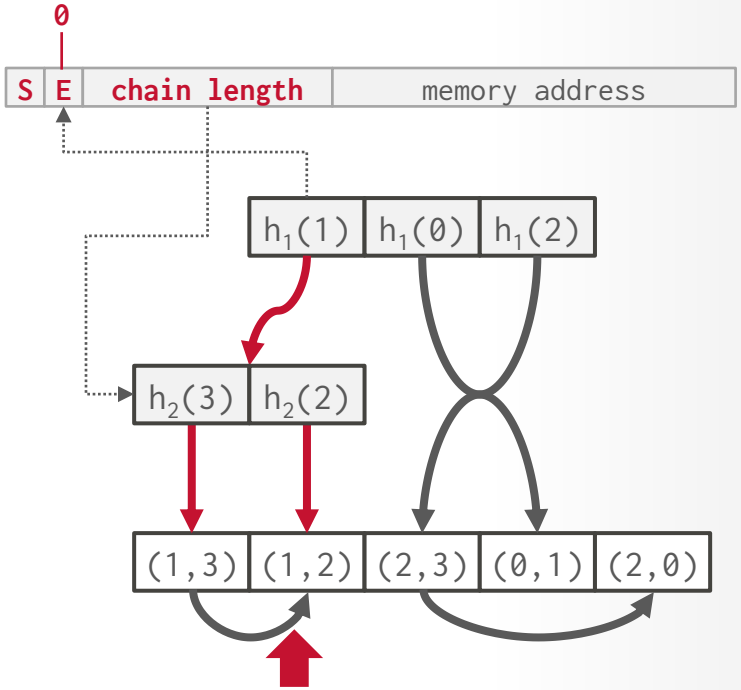
Optimization: Only materialize inner nodes in the trie when they are accessed during the probe phase.
→ Must always create root node.



HASH TRIE: LAZY CHILD EXPANSION

Depending on the selectivity of the intersection operation, the DBMS never accesses many inner nodes.

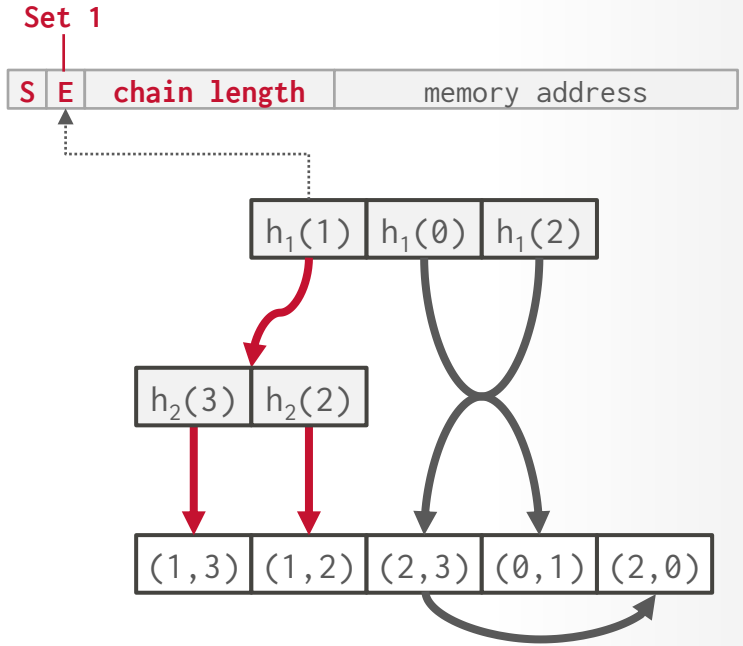
Optimization: Only materialize inner nodes in the trie when they are accessed during the probe phase.
→ Must always create root node.



HASH TRIE: LAZY CHILD EXPANSION

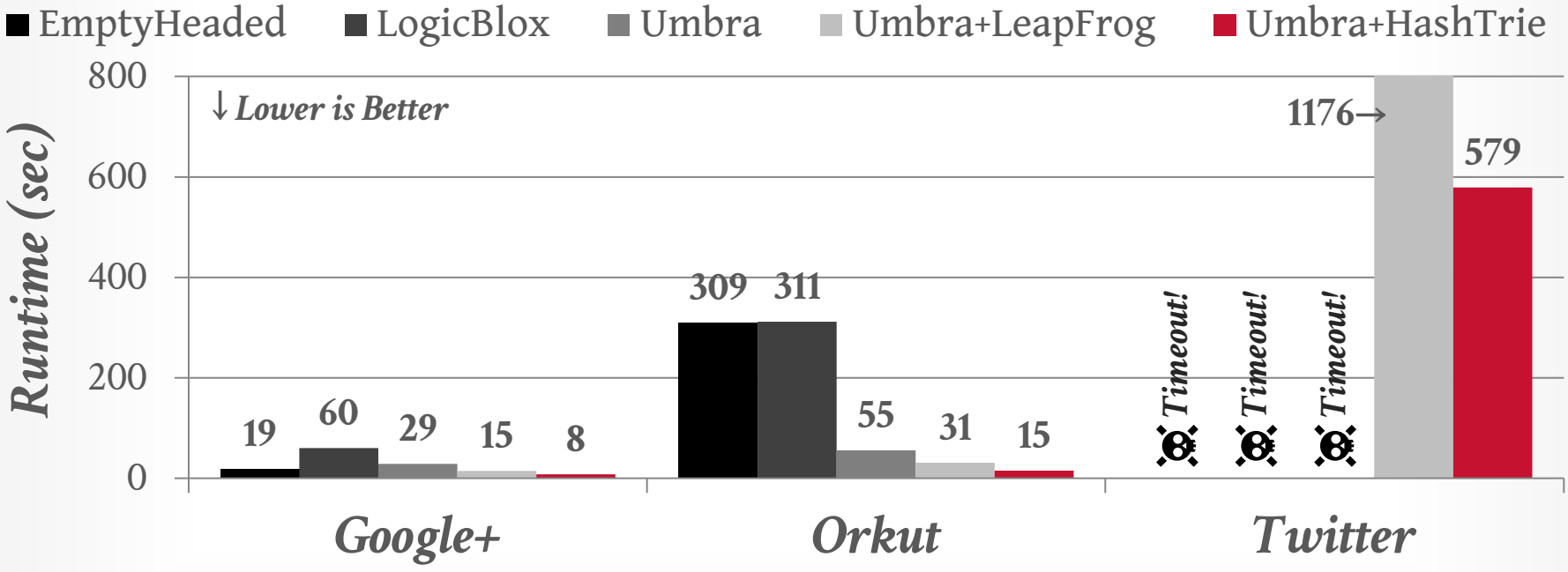
Depending on the selectivity of the intersection operation, the DBMS never accesses many inner nodes.

Optimization: Only materialize inner nodes in the trie when they are accessed during the probe phase.
→ Must always create root node.



WCOJ COMPARISON

2× Intel Xeon CPU E5-2680v4 (28 cores)
3-Clique Query on Graph Data Sets



Source: Michael Freitag

OBSERVATION

Multi-way joins are slower than binary joins if the query's intermediate results are not larger than its inputs. Choosing when to use a WCOJ algorithm is non-trivial because it relies on cost model estimates.

Umbra extended their optimizer to use heuristics to decide whether to use binary join vs. WCOJ.

OTHER OPTIMIZATIONS

Beyond multi-way joins, there are other optimizations a DBMS can utilize to execute multi-join queries more efficiently.

These are beneficial for both OLAP queries and graph pattern queries.

DUCKPGQ: EFFICIENT PROPERTY GRAPH QUERIES IN AN ANALYTICAL RDBMS
CIDR 2023

FACTORIZATION

A variant of late materialization where the DBMS maintains a single entry for duplicate tuples with a counter for number of occurrences.

Requires rewriting operators to support processing over factorized intermediate results.

$R \bowtie S$

a	b	c
0	0	0
0	0	0
0	0	1
0	0	2
1	1	0
2	2	0
0	0	2
1	1	0
2	2	0
0	0	1
0	0	2

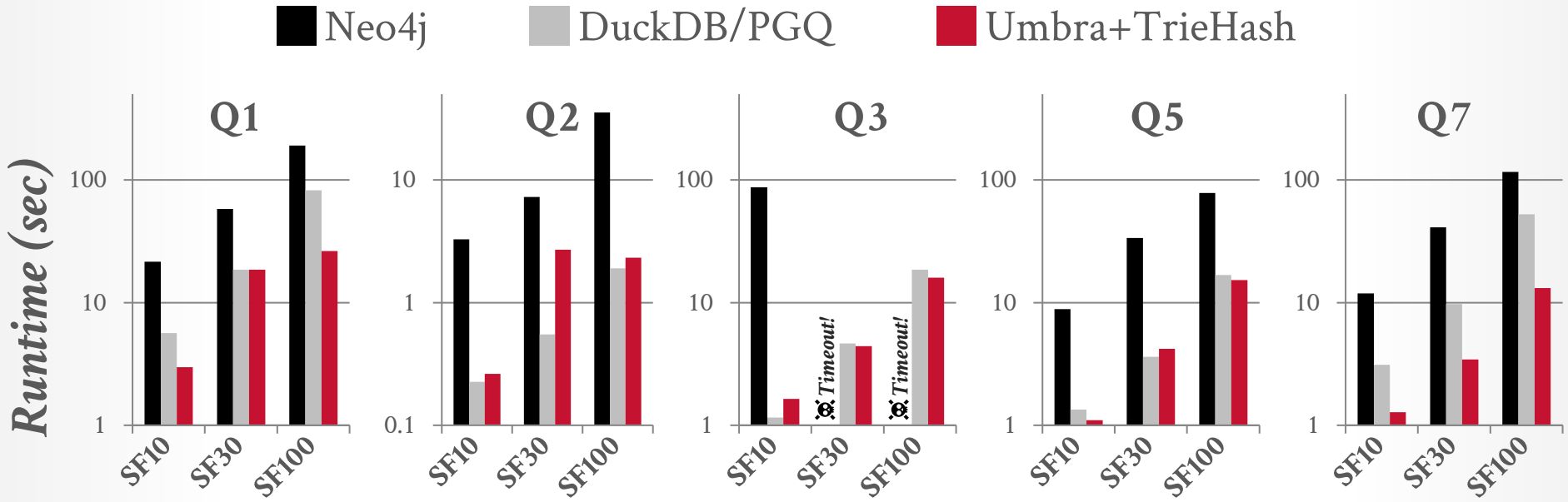
$R \bowtie S$

ctr	a	b	c
2	0	0	0
2	0	0	1
3	0	0	2
2	1	1	0
2	2	2	0

AGGREGATION AND ORDERING IN FACTORISED DATABASES
VLDB 2013

LSQB PATTERN MATCHING

Intel Xeon Platinum 8375C (48 Cores)
Labeled Subgraph Query Benchmark



Source: Daniel ten Wolde

PARTING THOUGHTS

Multi-way joins are an active area of research.

In the next decade they could be the default choice for supporting joins in a relational DBMS.

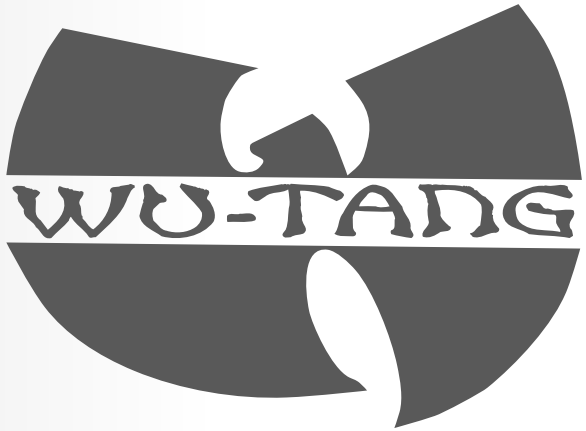
This will prevent graph-oriented DBMSs from overtaking relational DBMSs.

NEXT CLASS

Project Update #1 Presentations

→ Email me updated design document + slides PDF before class with "15-721 Status Update" in subject.

We will record the presentations (private) so that Will and I can go over them again and provide each group with feedback.



Andy's Street Tips for

DATABASE SYSTEM PROFILING

MOTIVATION

Consider a program with functions `foo` and `bar`.

How can we speed it up with only a debugger ?

→ Randomly pause it during execution

→ Collect the function call stack

RANDOM PAUSE METHOD

Consider this scenario

→ Collected 10 call stack samples

→ Say 6 out of the 10 samples were in `foo`

What percentage of time was spent in `foo`?

→ Roughly 60% of the time was spent in `foo`

→ Accuracy increases with # of samples

AMDAHL'S LAW

Say we optimized `foo` to run two times faster

What's the expected overall speedup ?

→ 60% of time spent in `foo` drops in half

→ 40% of time spent in `bar` unaffected

By Amdahl's law, overall speedup = $\frac{1}{\frac{p}{s}+(1-p)}$

→ p = percentage of time spent in optimized task

→ s = speed up for the optimized task

→ Overall speedup = $\frac{1}{\frac{0.6}{2}+0.4}$ = 1.4× faster

PROFILING TOOLS FOR REAL

Choice #1: Valgrind

→ Heavyweight binary instrumentation framework with different tools to measure different events.

Choice #2: Perf

→ Lightweight tool that uses hardware counters to capture events during execution.

CHOICE #1: VALGRIND

Instrumentation framework for building dynamic analysis tools.

→ **`memcheck`**: a memory error detector
→ **`callgrind`**: a call-graph generating profiler
→ **`massif`**: memory usage tracking.

KCACHEGRIND

Using callgrind to profile the target benchmark and the overall DBMS in general:

```
$ export TERRIER_BENCHMARK_THREADS=16
$ valgrind --tool=callgrind --trace-children=yes
./relwithdebinfo/slot_iterator_benchmark
```

KCACHEGRIND

Using callgrind to profile the target benchmark and the overall DBMS in general:

```
$ valgrind --tool=callgrind --trace-children=yes
./relwithdebinfo/slot_iterator_benchmark
```

Profile data visualization tool:

```
$ kcachegrind callgrind.out.12345
```

/home/pavlo/Documents/Peloton/Github/terrier/build/callgrind.out.25460 [./relwithdebinfo/concurrent_read_benchmark] — KCachegrind

File View Go Settings Help

Open... | Back | Forward | Up | Relative | Cycle Detection | Relative to Parent | Shorten Templates | Instruction Fetch

Flat Profile

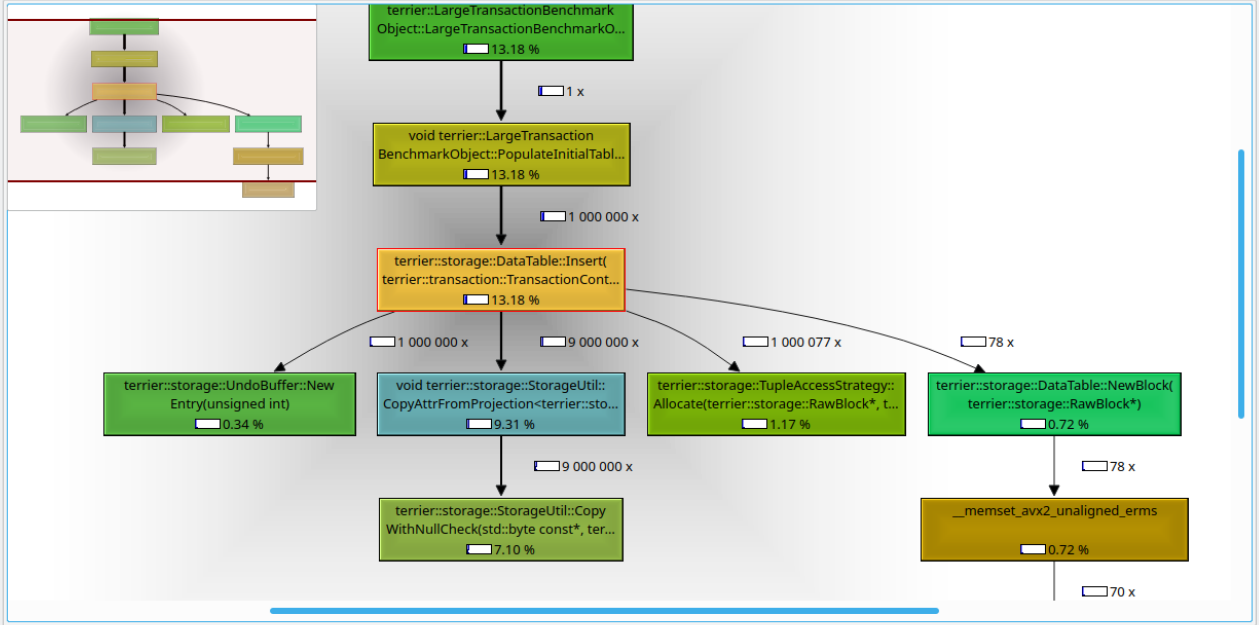
Search: (No Grouping)

Incl.	Self	Called	Function	Local
63.32	0.00	(0)	0x0000000000001090	ld-2.2
63.30	0.00	1	_start	concu
63.30	0.00	1	(below main)	libc-2
63.28	0.00	1	main	concu
63.28	0.00	1	benchmark::RunSpecifiedB...	concu
63.28	0.00	1	benchmark::RunSpecifiedB...	concu
63.28	0.00	1	terrier::ConcurrentReadBe...	concu
63.27	0.00	1	terrier::LargeTransactionBe...	concu
63.27	10.36	1	void terrier::LargeTransacti...	concu
36.68	0.00	21	start_thread	libpth
36.68	0.00	33	0x00000000000bd570	libstd
35.99	35.99	72 000 000	unsigned char std::uniform...	concu
32.73	0.00	32	std::thread::_State_impl<st...	concu
32.73	0.00	16	std::_Function_handler<voi...	concu
32.73	0.16	16	std::_Function_handler<voi...	concu
26.49	2.80	1 000 000	terrier::LargeTransactionBe...	concu
24.39	0.00	12	clone'2	libc-2
24.39	0.00	12	start_thread'2	libpth
13.18	1.38	1 000 000	terrier::storage::DataTable::...	concu
13.13	0.37	1 000 000	std::_Function_handler<voi...	concu
10.22	0.01	1 000 000	terrier::storage::DataTable::...	concu
10.22	1.01	1 000 000	bool terrier::storage::DataT...	concu
9.31	2.21	9 000 000	void terrier::storage::Stora...	concu
8.44	3.79	9 000 000	void terrier::storage::Stora...	concu
7.10	7.10	9 000 000	terrier::storage::StorageUtil...	concu
6.33	0.14	8 010 333	operator delete(void*)	libstd
6.19	6.19	8 010 990	free	libc-2
5.32	0.84	8 010 335	operator new(unsigned long)	libstd
4.76	0.28	1 000 000	terrier::RandomWorkloadTr...	concu

terrier::storage::DataTable::Insert(terrier::transaction::TransactionContext*, terrier::storage::ProjectedRow const&)

Types | Callers | All Callers | Callee Map | Source Code

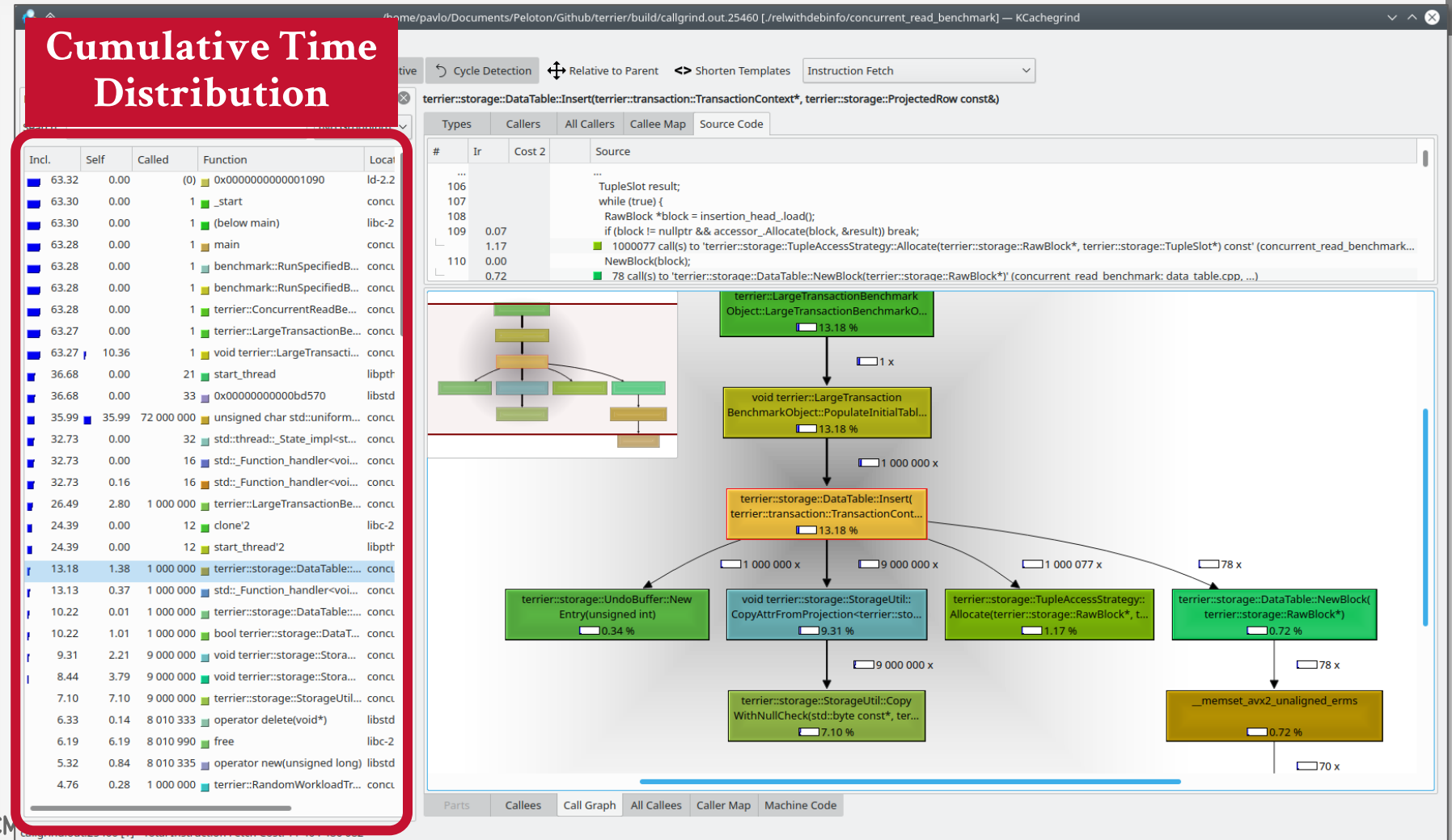
#	Ir	Cost 2	Source
...			...
106			TupleSlot result;
107			while (true) {
108			RawBlock *block = insertion_head_.load();
109	0.07		if (block != nullptr && accessor_.Allocate(block, &result)) break;
	1.17		1000077 call(s) to 'terrier::storage::TupleAccessStrategy::Allocate(terrier::storage::RawBlock*, terrier::storage::TupleSlot*) const' (concurrent_read_benchmark...
110	0.00		NewBlock(block);
	0.72		78 call(s) to 'terrier::storage::DataTable::NewBlock(terrier::storage::RawBlock*)' (concurrent_read_benchmark: data_table.cpp, ...)



Parts | Callees | Call Graph | All Callees | Caller Map | Machine Code

callgrind.out.25460 [1] - Total Instruction Fetch Cost: 11 404 486 082

Cumulative Time Distribution



Cumulative Time Distribution

/home/pavlo/Documents/Peloton/Github/terrier/build/callgrind.out.25460 [./relwithdebinfo/concurrent_read_benchmark] — KCachegrind

Cycle Detection | Relative to Parent | Shorten Templates | Instruction Fetch

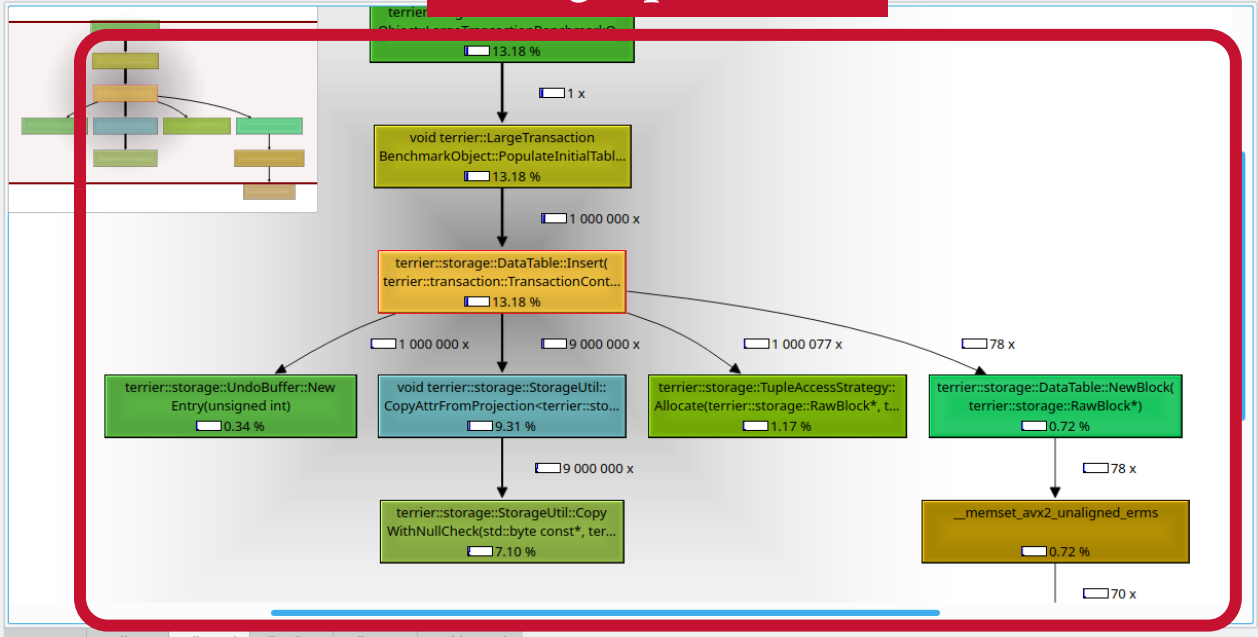
terrier::storage::DataTable::Insert(terrier::transaction::TransactionContext*, terrier::storage::ProjectedRow const&)

Types | Callers | All Callers | Callee Map | Source Code

#	Ir	Cost 2	Source
106			TupleSlot result;
107			while (true) {
108			RawBlock *block = insertion_head_.load();
109	0.07		if (block != nullptr && accessor_.Allocate(block, &result)) break;
	1.17		1000077 call(s) to 'terrier::stora... ...rage::TupleSlot*) const' (concurrent_read_benchmark...
110	0.00		NewBlock(block);
	0.72		78 call(s) to 'terrier::storage::D... ...mark: data_table.cpp, ...)

Incl.	Self	Called	Function	Local
63.32	0.00	(0)	0x0000000000001090	ld-2.2
63.30	0.00	1	_start	concu
63.30	0.00	1	(below main)	libc-2
63.28	0.00	1	main	concu
63.28	0.00	1	benchmark::RunSpecifiedB...	concu
63.28	0.00	1	benchmark::RunSpecifiedB...	concu
63.28	0.00	1	terrier::ConcurrentReadBe...	concu
63.27	0.00	1	terrier::LargeTransactionBe...	concu
63.27	10.36	1	void terrier::LargeTransacti...	concu
36.68	0.00	21	start_thread	libpth
36.68	0.00	33	0x00000000000bd570	libstd
35.99	35.99	72 000 000	unsigned char std::uniform...	concu
32.73	0.00	32	std::thread::_State_impl<st...	concu
32.73	0.00	16	std::_Function_handler<voi...	concu
32.73	0.16	16	std::_Function_handler<voi...	concu
26.49	2.80	1 000 000	terrier::LargeTransactionBe...	concu
24.39	0.00	12	clone'2	libc-2
24.39	0.00	12	start_thread'2	libpth
13.18	1.38	1 000 000	terrier::storage::DataTable::...	concu
13.13	0.37	1 000 000	std::_Function_handler<voi...	concu
10.22	0.01	1 000 000	terrier::storage::DataTable::...	concu
10.22	1.01	1 000 000	bool terrier::storage::DataT...	concu
9.31	2.21	9 000 000	void terrier::storage::Stora...	concu
8.44	3.79	9 000 000	void terrier::storage::Stora...	concu
7.10	7.10	9 000 000	terrier::storage::StorageUtil...	concu
6.33	0.14	8 010 333	operator delete(void*)	libstd
6.19	6.19	8 010 990	free	libc-2
5.32	0.84	8 010 335	operator new(unsigned long)	libstd
4.76	0.28	1 000 000	terrier::RandomWorkloadTr...	concu

Callgraph View



Parts | Callees | Call Graph | All Callees | Caller Map | Machine Code

CHOICE #2: PERF

Tool for using the performance counters subsystem in Linux.

→ **-e** = sample the event cycles at the user level only

→ **-c** = collect a sample every 2000 occurrences of event

```
$ perf record -e cycles:u -c 2000
./relwithdebinfo/slot_iterator_benchmark
```

Uses counters for tracking events

→ On counter overflow, the kernel records a sample

→ Sample contains info about program execution

PERF VISUALIZATION

We can also use **perf** to visualize the generated profile for our application.

```
$ perf report
```

There are also third-party visualization tools:
→ Hotspot

PERF VISUALIZATION

```
Samples: 9M of event 'cycles:u', Event count (approx.): 18388130000
Overhead  Command          Shared Object               Symbol
  17.89%  concurrent_read  concurrent_read_benchmark  [.] _ZN7terrier7storage11StorageUtil17CopyWithNullCheckINS0_12Projecte
   9.41%  concurrent_read  concurrent_read_benchmark  [.] _ZN7terrier11transaction18TransactionManager6CommitEPNS0_18Transac
   9.36%  concurrent_read  concurrent_read_benchmark  [.] _ZN7terrier11transaction18TransactionManager16BeginTransactionEPNS
   8.10%  concurrent_read  concurrent_read_benchmark  [.] _ZNSt24uniform_int_distributionIhEclISt26linear_congruential_engin
   5.27%  concurrent_read  concurrent_read_benchmark  [.] _ZN7terrier7storage11StorageUtil22CopyAttrIntoProjectionINS0_12Pro
   3.53%  concurrent_read  libc-2.27.so               [.] _int_malloc
   3.28%  concurrent_read  libc-2.27.so               [.] __sched_yield
   3.08%  concurrent_read  libc-2.27.so               [.] cfree@GLIBC_2.2.5
   3.06%  concurrent_read  concurrent_read_benchmark  [.] _ZNSt17_Function_handlerIFvvEZN7terrier31LargeTransactionBenchmark
   2.87%  concurrent_read  concurrent_read_benchmark  [.] _ZNK7terrier7storage9DataTable24AtomicallyReadVersionPtrENS0_9Tupl
   2.72%  concurrent_read  concurrent_read_benchmark  [.] _ZNKSt10_HashtableIN7terrier6common15StrongTypeAliasINS0_11transac
   2.45%  concurrent_read  concurrent_read_benchmark  [.] _ZN7terrier7storage16GarbageCollector18ProcessUnlinkQueueEv
   1.86%  concurrent_read  concurrent_read_benchmark  [.] _ZN7terrier7storage11StorageUtil17CopyWithNullCheckEPKSt4byteRKNS0
   1.74%  concurrent_read  libtbb.so.2                [.] 0x0000000000018ac4
   1.58%  concurrent_read  libc-2.27.so               [.] malloc
   1.20%  concurrent_read  concurrent_read_benchmark  [.] _ZNSt10_HashtableIN7terrier6common15StrongTypeAliasINS0_11transact
   0.99%  concurrent_read  libc-2.27.so               [.] __memset_avx2_unaligned_erms
   0.98%  concurrent_read  concurrent_read_benchmark  [.] _ZNSt17_Function_handlerIFvjEZN7terrier31LargeTransactionBenchmark
   0.90%  concurrent_read  libtbb.so.2                [.] 0x00000000000185cb
   0.89%  concurrent_read  concurrent_read_benchmark  [.] _ZN7terrier31LargeTransactionBenchmarkObject22SimulateOneTransacti
   0.83%  concurrent_read  concurrent_read_benchmark  [.] _ZSt18generate_canonicalIdLm53ESt26linear_congruential_engineImLm1
   0.72%  concurrent_read  concurrent_read_benchmark  [.] _ZN7terrier11transaction18TransactionManager9LogCommitEPNS0_18Tran
   0.70%  concurrent_read  concurrent_read_benchmark  [.] _ZN7terrier31LargeTransactionBenchmarkObject20PopulateInitialTable
   0.70%  concurrent_read  libtbb.so.2                [.] 0x0000000000018ac6
   0.68%  concurrent_read  [kernel]                   [k] 0xfffffe000005e000
   0.57%  concurrent_read  concurrent_read_benchmark  [.] _ZNK7terrier7storage9DataTable16SelectIntoBufferINS0_12ProjectedRo
   0.56%  concurrent_read  [kernel]                   [k] 0xfffffe00000e2000
Cannot load tips.txt file, please install perf!
```

PERF VISUALIZATION

```
Samples: 9M of event 'cycles:u', Event count (approx.): 18388130000
Overhead  Command          Shared Object              Symbol
  17.89%  concurrent_read  concurrent_read_benchmark  [.] _ZN7terrier7storage11StorageUtil17CopyWithNullCheckINS0_12Projecte
   9.41%  concurrent_read  concurrent_read_benchmark  [.] _ZN7terrier11transaction18TransactionManager6CommitEPNS0_18Transac
   9.36%  concurrent_read  concurrent_read_benchmark  [.] _ZN7terrier11transaction18TransactionManager16BeginTransactionEPNS
   8.10%  concurrent_read  concurrent_read_benchmark  [.] _ZNSt24uniform_int_distributionIhEclISt26linear_congruential_engin
   5.27%  concurrent_read  concurrent_read_benchmark  [.] _ZN7terrier7storage11StorageUtil22CopyAttrIntoProjectionINS0_12Pro
   3.53%  concurrent_read  libc-2.27.so               [.] _int_malloc
   3.28%  concurrent_read  libc-2.27.so               [.] __sched_yield
   3.08%  concurrent_read  libc-2.27.so               [.] cfree@GLIBC_2.2.5
   3.06%  concurrent_read  concurrent_read_benchmark  [.] _ZNSt17_Function_handlerIFvvEZN7terrier31LargeTransactionBenchmark
   2.87%  concurrent_read  concurrent_read_benchmark  [.] _ZNK7terrier7storage9DataTable24AtomicallyReadVersionPtrENS0_9Tupl
   2.72%  concurrent_read  concurrent_read_benchmark  [.] _ZNKSt10_HashtableIN7terrier6common15StrongTypeAliasINS0_11transac
   2.45%  concurrent_read  concurrent_read_benchmark  [.] _ZN7terrier7storage16GarbageCollector18ProcessUnlinkQueueEv
   1.86%  concurrent_read  concurrent_read_benchmark  [.] _ZN7terrier7storage11StorageUtil17CopyWithNullCheckEPKSt4byteRKNS0
   1.74%  concurrent_read  libtbb.so.2                [.] 0x0000000000018ac4
   1.58%  concurrent_read  libc-2.27.so               [.] malloc
   1.20%  concurrent_read  concurrent_read_benchmark  [.] _ZNSt10_HashtableIN7terrier6common15StrongTypeAliasINS0_11transact
   0.99%  concurrent_read  libc-2.27.so               [.] __memset_avx2_unaligned_erms
   0.98%  concurrent_read  concurrent_read_benchmark  [.] _ZNSt17_Function_handlerIFvjEZN7terrier31LargeTransactionBenchmark
   0.90%  concurrent_read  libtbb.so.2                [.] 0x00000000000185cb
   0.89%  concurrent_read  concurrent_read_benchmark  [.] _ZN7terrier31LargeTransactionBenchmarkObject22SimulateOneTransacti
   0.83%  concurrent_read  concurrent_read_benchmark  [.] _ZSt18generate_canonicalIdLm53ESt26linear_congruential_engineImLm1
   0.72%  concurrent_read  concurrent_read_benchmark  [.] _ZN7terrier11transaction18TransactionManager9LogCommitEPNS0_18Tran
          concurrent_read  concurrent_read_benchmark  [.] _ZN7terrier31LargeTransactionBenchmarkObject20PopulateInitialTable
          libtbb.so.2                                 [.] 0x0000000000018ac6
          [kernel]                                    [k] 0xfffffe000005e000
          concurrent_read  concurrent_read_benchmark  [.] _ZNK7terrier7storage9DataTable16SelectIntoBufferINS0_12ProjectedRo
          [kernel]                                    [k] 0xfffffe00000e2000
Cannot load tips.txt file, please install perf!
```

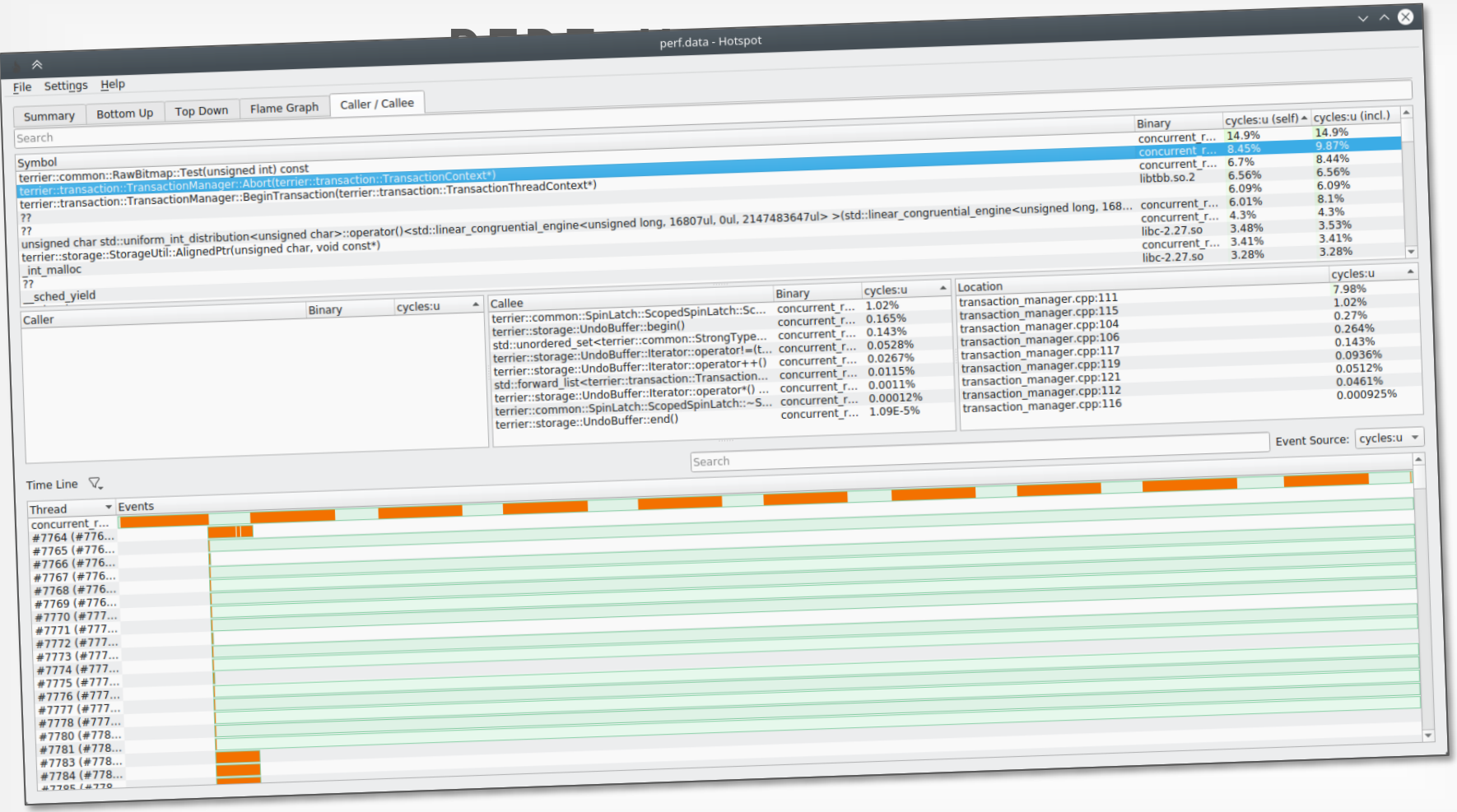
Cumulative Event Distribution

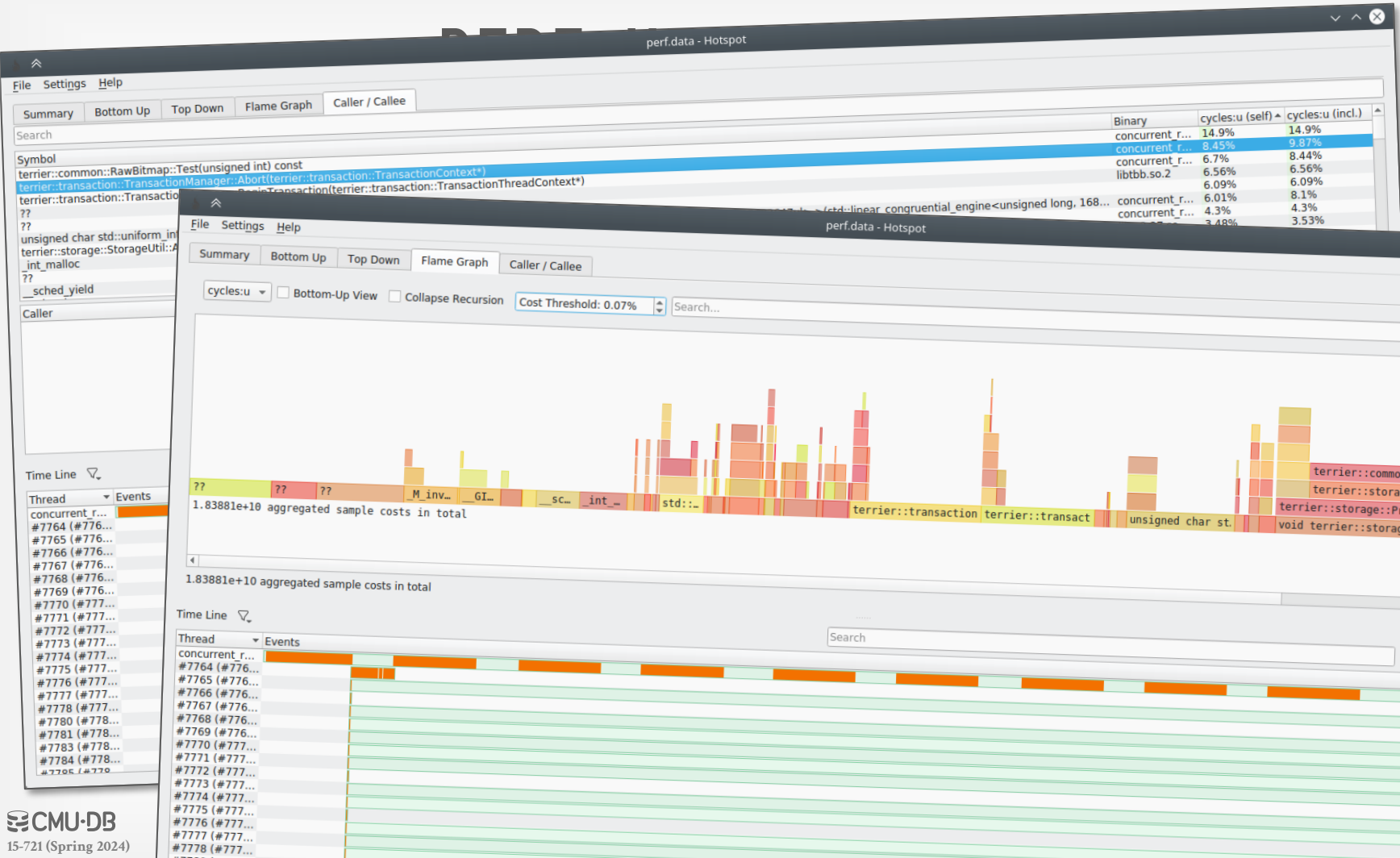
PERF VISUALIZATION

We can also use **perf** to visualize the generated profile for our application.

```
$ perf report
```

There are also third-party visualization tools:
→ Hotspot


perf.data - Hotspot
File Settings Help
Summary Bottom Up Top Down Flame Graph Caller / Callee
Search
Symbol
terrier::common::RawBitmap::Test(unsigned int) const
terrier::transaction::TransactionManager::Abort(terrier::transaction::TransactionContext*)
terrier::transaction::TransactionManager::BeginTransaction(terrier::transaction::TransactionThreadContext*)
??
??
unsigned char std::uniform_int_distribution<unsigned char>::operator()<std::linear_congruential_engine<unsigned long, 16807ul, 0ul, 2147483647ul> >(std::linear_congruential_engine<unsigned long, 168...
terrier::storage::StorageUtil::AlignedPtr(unsigned char, void const*)
_int_malloc
??
__sched_yield
Binary
cycles:u (self)
cycles:u (incl.)
14.9% 14.9%
8.45% 9.87%
6.7% 8.44%
6.56% 6.56%
6.09% 6.09%
6.01% 8.1%
4.3% 4.3%
3.48% 3.53%
3.41% 3.41%
3.28% 3.28%
Caller
Binary
cycles:u
Callee
Binary
cycles:u
terrier::common::SpinLatch::ScopedSpinLatch::Sc... 1.02%
terrier::storage::UndoBuffer::begin() 0.165%
std::unordered_set<terrier::common::StrongType... 0.143%
terrier::storage::UndoBuffer::Iterator::operator!=(t... 0.0528%
terrier::storage::UndoBuffer::Iterator::operator++() 0.0267%
std::forward_list<terrier::transaction::Transaction... 0.0115%
terrier::storage::UndoBuffer::Iterator::operator*() ... 0.0011%
terrier::common::SpinLatch::ScopedSpinLatch::~S... 0.00012%
terrier::storage::UndoBuffer::end() 1.09E-5%
Location
cycles:u
transaction_manager.cpp:111 7.98%
transaction_manager.cpp:115 1.02%
transaction_manager.cpp:104 0.27%
transaction_manager.cpp:106 0.264%
transaction_manager.cpp:117 0.143%
transaction_manager.cpp:119 0.0936%
transaction_manager.cpp:121 0.0512%
transaction_manager.cpp:112 0.0461%
transaction_manager.cpp:116 0.000925%
Search
Event Source: cycles:u
Time Line
Thread
Events
concurrent_r...
#7764 (#776...
#7765 (#776...
#7766 (#776...
#7767 (#776...
#7768 (#776...
#7769 (#776...
#7770 (#777...
#7771 (#777...
#7772 (#777...
#7773 (#777...
#7774 (#777...
#7775 (#777...
#7776 (#777...
#7777 (#777...
#7778 (#777...
#7780 (#778...
#7781 (#778...
#7783 (#778...
#7784 (#778...


perf.data - Hotspot
File Settings Help
Summary Bottom Up Top Down Flame Graph Caller / Callee
Search
Symbol
terrier::common::RawBitmap::Test(unsigned int) const
terrier::transaction::TransactionManager::Abort(terrier::transaction::TransactionContext*)
terrier::transaction::TransactionManager::BeginTransaction(terrier::transaction::TransactionThreadContext*)
??
??
unsigned char std::uniform_int
terrier::storage::StorageUtil::A
_int_malloc
??
__sched_yield
Binary
cycles:u (self)
cycles:u (incl.)
concurrent_r...
14.9%
14.9%
8.45%
9.87%
6.7%
8.44%
libtbb.so.2
6.56%
6.56%
6.09%
6.09%
6.01%
8.1%
4.3%
4.3%
3.53%
Caller
Time Line
Thread
Events
concurrent_r...
#7764 (#776...
#7765 (#776...
#7766 (#776...
#7767 (#776...
#7768 (#776...
#7769 (#776...
#7770 (#777...
#7771 (#777...
#7772 (#777...
#7773 (#777...
#7774 (#777...
#7775 (#777...
#7776 (#777...
#7777 (#777...
#7778 (#777...
#7780 (#778...
#7781 (#778...
#7783 (#778...
#7784 (#778...
perf.data - Hotspot
File Settings Help
Summary Bottom Up Top Down Flame Graph Caller / Callee
cycles:u
Bottom-Up View
Collapse Recursion
Cost Threshold: 0.07%
Search...
?? ?? ?? _M_inv... __GI... __sc... _int_... std::... terrier::transaction terrier::transact unsigned char st...
terrier::common::RawBitmap::Te
terrier::storage::ProjectedColu
terrier::storage::ProjectedColumns::
void terrier::storage::StorageUtil::(
1.83881e+10 aggregated sample costs in total
1.83881e+10 aggregated sample costs in total
Time Line
Thread
Events
Search
Event Source:
cycles:u
concurrent_r...
#7764 (#776...
#7765 (#776...
#7766 (#776...
#7767 (#776...
#7768 (#776...
#7769 (#776...
#7770 (#777...
#7771 (#777...
#7772 (#777...
#7773 (#777...
#7774 (#777...
#7775 (#777...
#7776 (#777...
#7777 (#777...
#7778 (#777...

PERF EVENTS

Supports several other events like:
→ L1-dcache-load-misses
→ branch-misses

To see a list of events:

```
$ perf list
```

Another usage example:

```
$ perf record -e cycles,LLC-load-misses -c 2000
./relwithdebinfo/slot_iterator_benchmark
```

REFERENCES

Valgrind

→ The Valgrind Quick Start Guide
→ Callgrind
→ Kcachegrind
→ Tips for the Profiling/Optimization process

Perf

→ Perf Tutorial
→ Perf Examples
→ Perf Analysis Tools