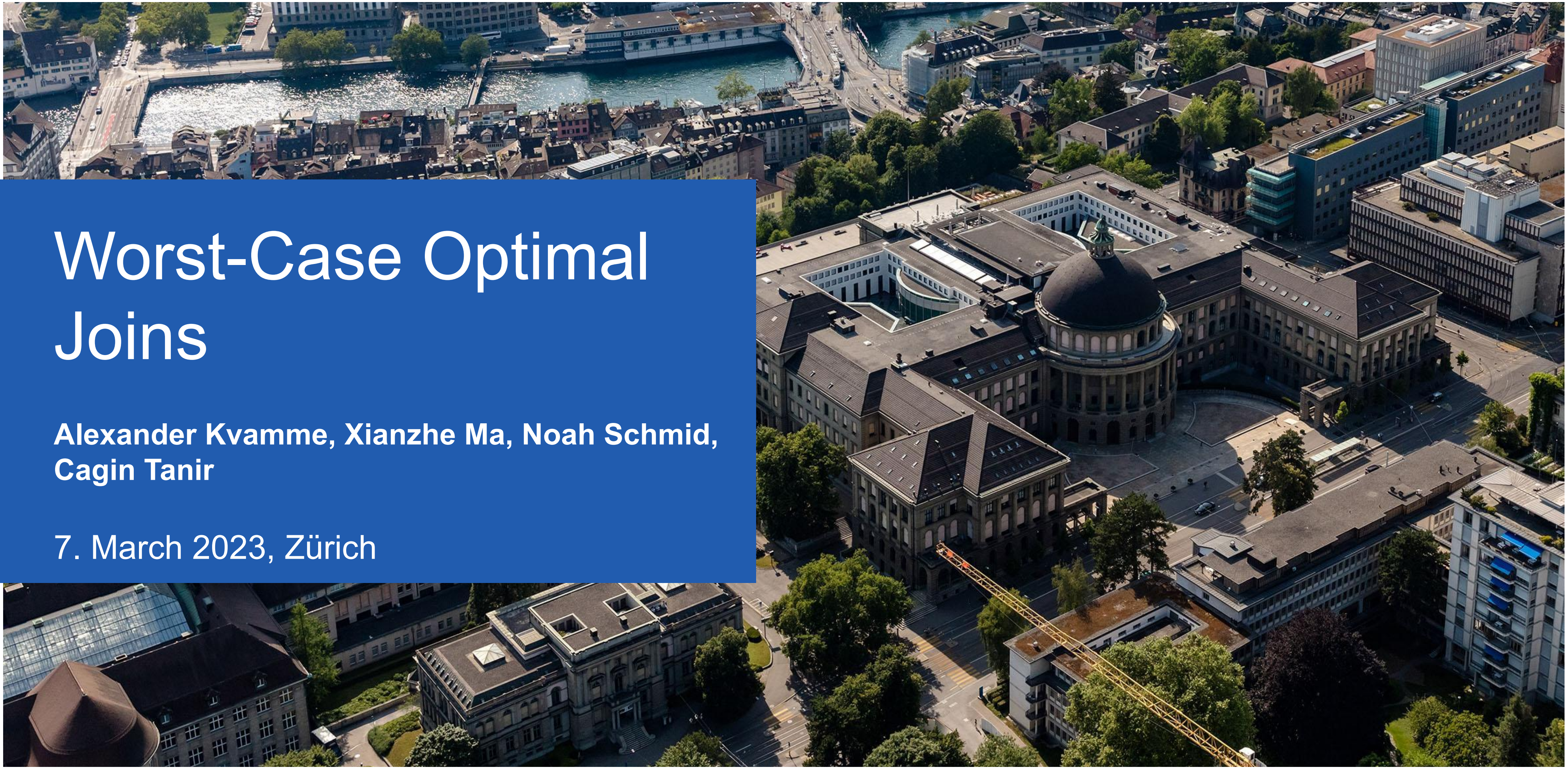
ETH zürich

Advanced Systems Lab



Worst-Case Optimal Joins

Alexander Kvamme, Xianzhe Ma, Noah Schmid, Cagin Tanir

7. March 2023, Zürich

Problem Statement

JOIN R1,R2,R3 on v1,v2,v3

R1	
v1	v2
0	1
1	2
1	3
2	0
2	3

R2	
v2	v3
0	1
1	2
1	3
2	0
2	3

R3	
v3	v1
0	1
1	2
1	3
2	0
2	3

Problem Statement

Intermediate Join		
v1	v2	v3
0	1	2
0	1	3
1	2	0
1	2	3
2	0	1

R3	
v3	v1
0	1
1	2
1	3
2	0
2	3

Problem Statement

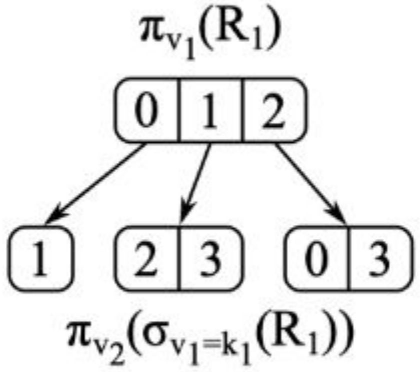
Join Result		
v1	v2	v3
0	1	2
1	2	3
2	0	1

Structure of the Algorithms

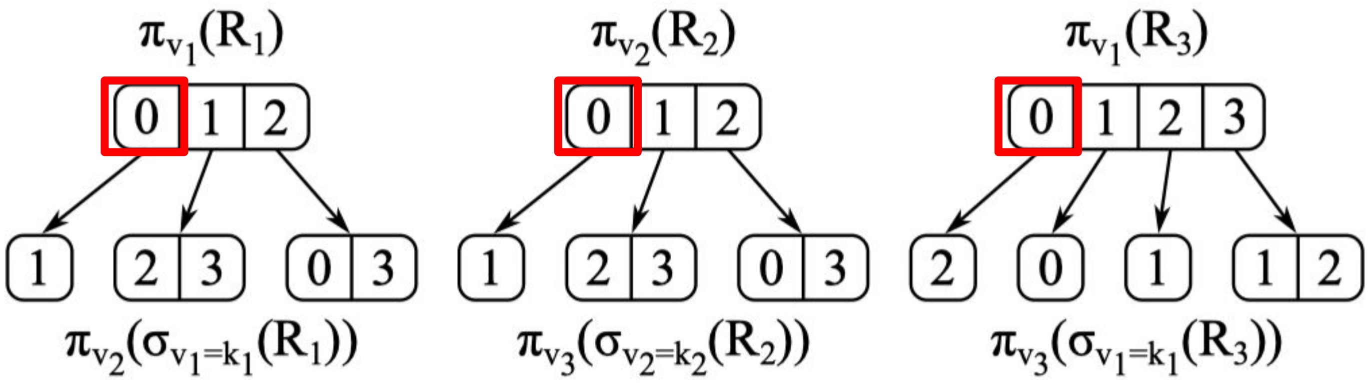
- Algorithm 2: Build Hash Trie
- Algorithm 3: Look for matching tuples
- Algorithm 3.5: Append joined tuples to result table

Algorithm 2

R1	
v1	v2
0	1
1	2
1	3
2	0
2	3

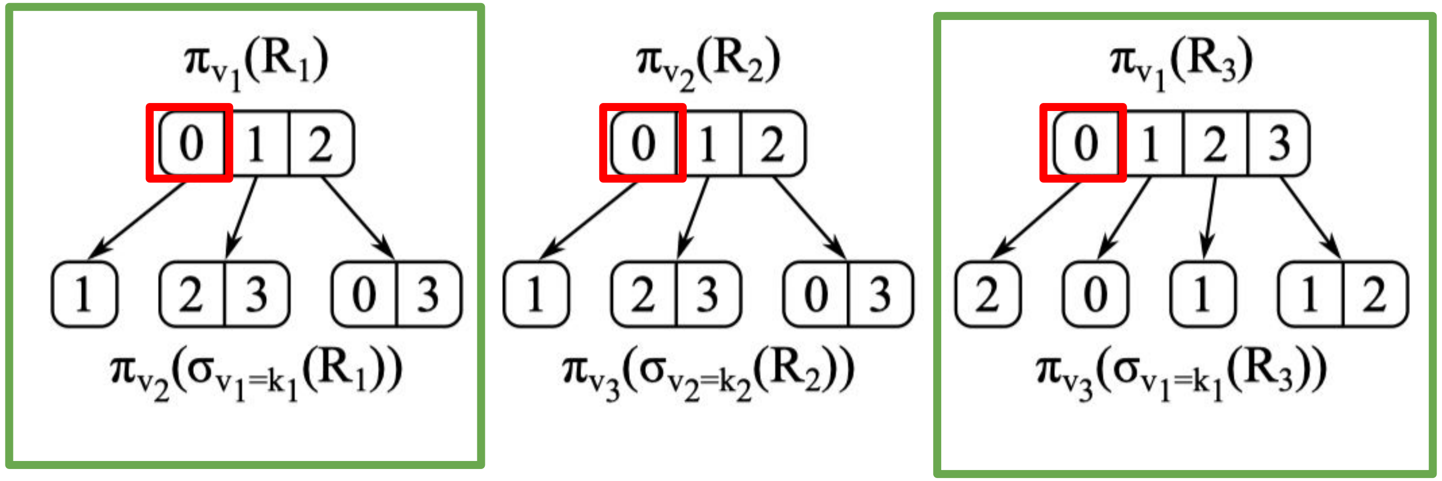


Algorithm 3



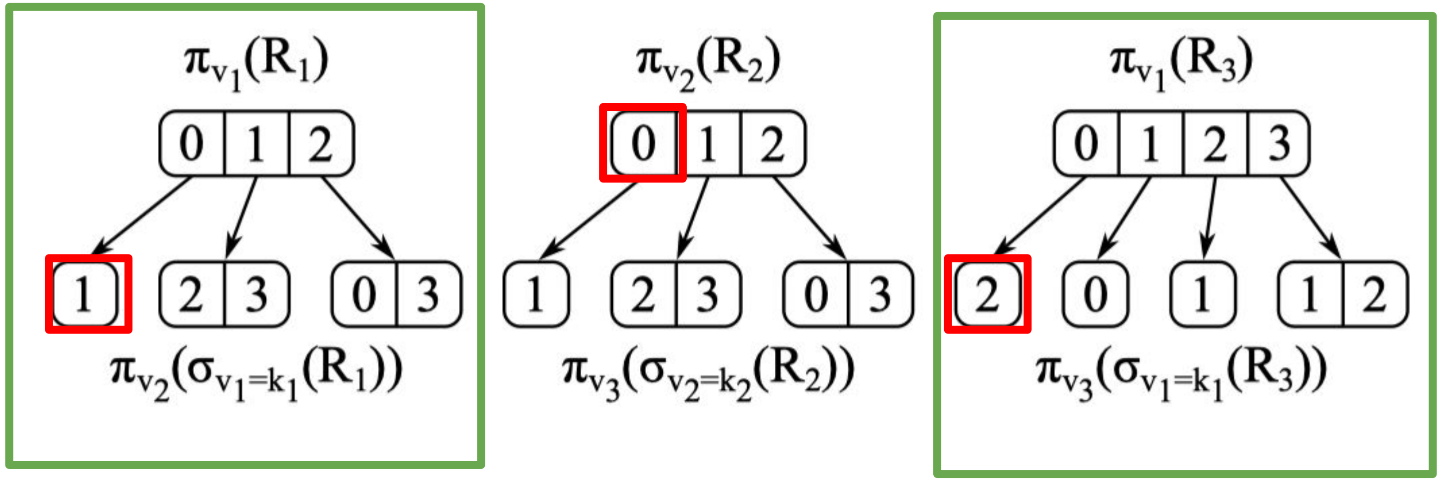
$\pi_{v_1}(R_1)$
0
1
2
1
2
3
0
3
$\pi_{v_2}(\sigma_{v_1=k_1}(R_1))$
$\pi_{v_2}(R_2)$
0
1
2
1
2
3
0
3
$\pi_{v_3}(\sigma_{v_2=k_2}(R_2))$
$\pi_{v_1}(R_3)$
0
1
2
3
2
0
1
1
2
$\pi_{v_3}(\sigma_{v_1=k_1}(R_3))$

Algorithm 3

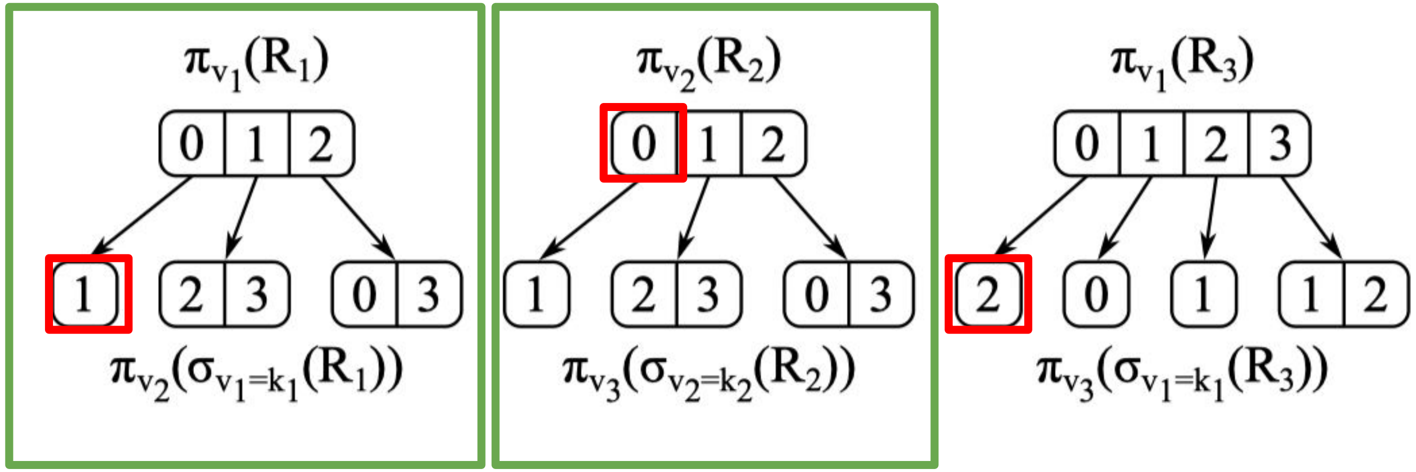


$\pi_{v_1}(R_1)$
0 1 2
1
2 3
0 3
$\pi_{v_2}(\sigma_{v_1=k_1}(R_1))$
$\pi_{v_2}(R_2)$
0 1 2
1
2 3
0 3
$\pi_{v_3}(\sigma_{v_2=k_2}(R_2))$
$\pi_{v_1}(R_3)$
0 1 2 3
2
0
1
1 2
$\pi_{v_3}(\sigma_{v_1=k_1}(R_3))$

Algorithm 3

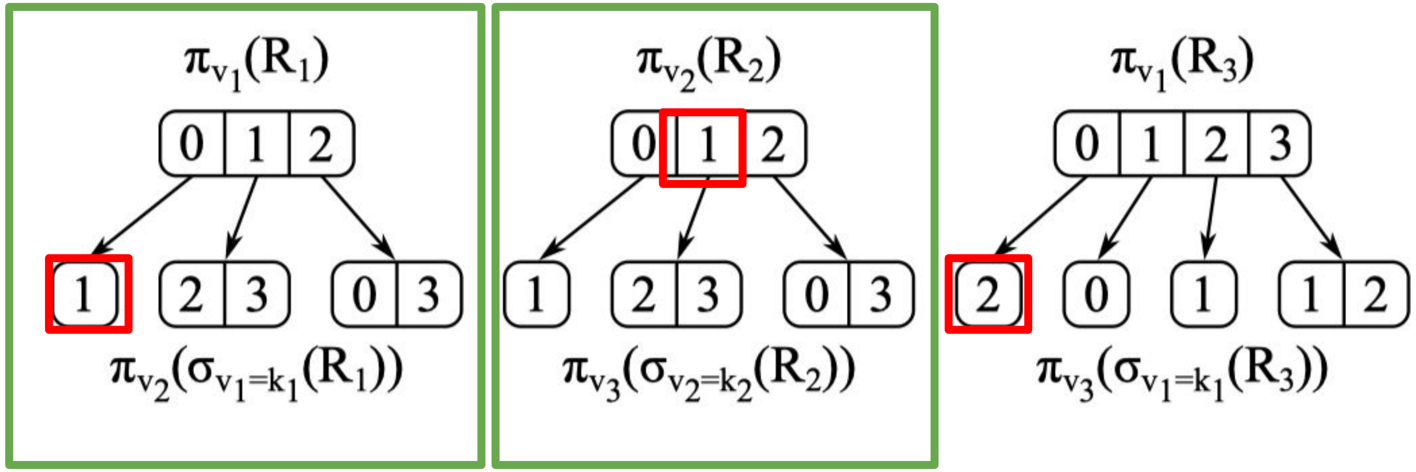

$\pi_{v_1}(R_1)$
0 1 2
1
2 3
0 3
$\pi_{v_2}(\sigma_{v_1=k_1}(R_1))$
$\pi_{v_2}(R_2)$
0 1 2
1
2 3
0 3
$\pi_{v_3}(\sigma_{v_2=k_2}(R_2))$
$\pi_{v_1}(R_3)$
0 1 2 3
2
0
1
1 2
$\pi_{v_3}(\sigma_{v_1=k_1}(R_3))$

Algorithm 3

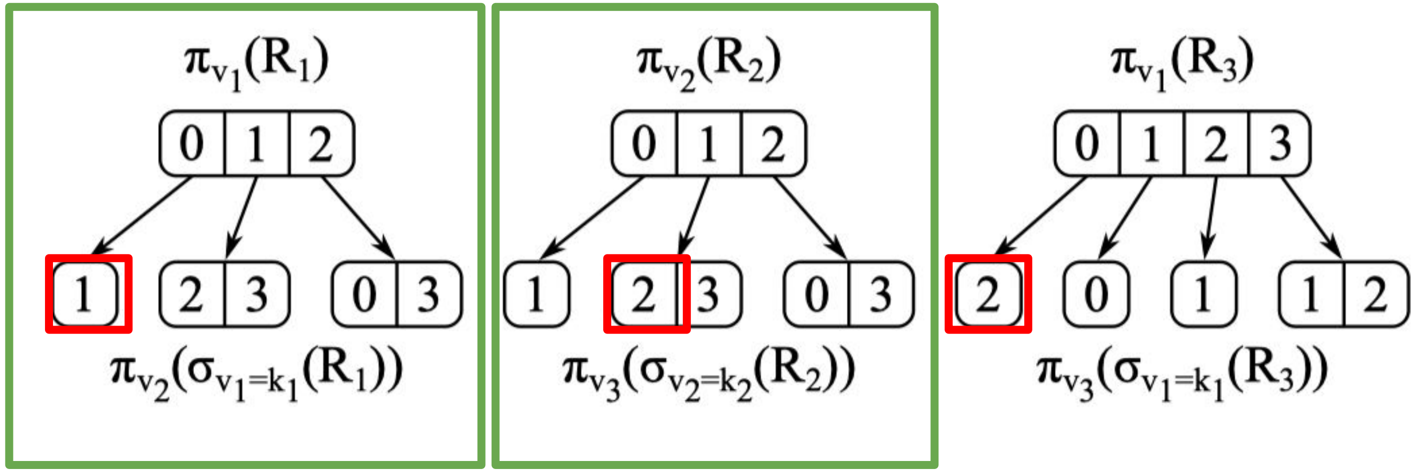


$\pi_{v_1}(R_1)$
0 1 2
1
2 3
0 3
$\pi_{v_2}(\sigma_{v_1=k_1}(R_1))$
$\pi_{v_2}(R_2)$
0 1 2
1
2 3
0 3
$\pi_{v_3}(\sigma_{v_2=k_2}(R_2))$
$\pi_{v_1}(R_3)$
0 1 2 3
2
0
1
1 2
$\pi_{v_3}(\sigma_{v_1=k_1}(R_3))$

Algorithm 3

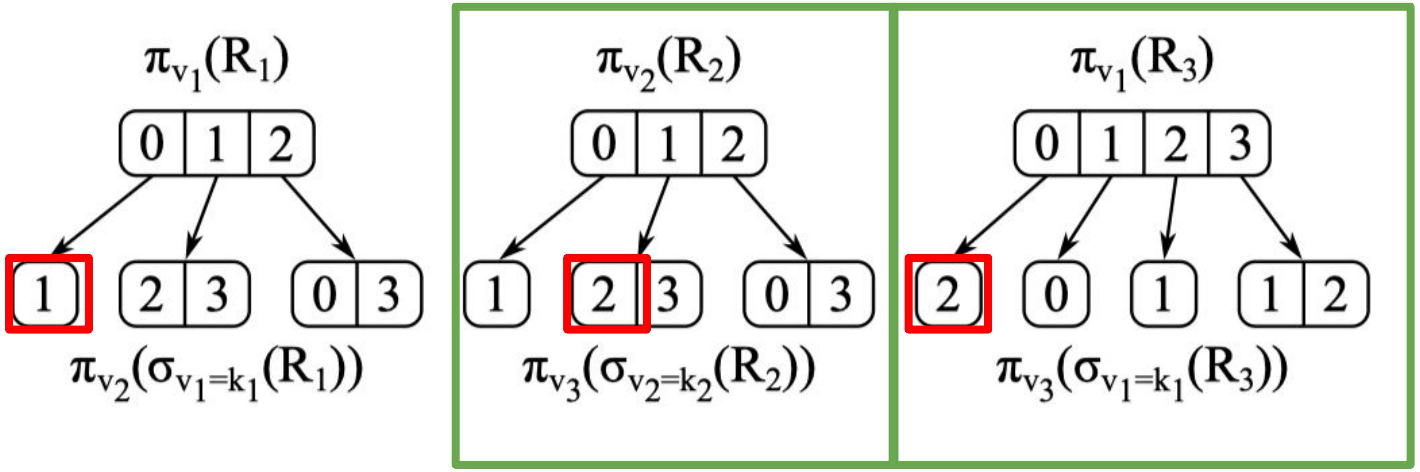

$\pi_{v_1}(R_1)$
0 1 2
1
2 3
0 3
$\pi_{v_2}(\sigma_{v_1=k_1}(R_1))$
$\pi_{v_2}(R_2)$
0 1 2
1
2 3
0 3
$\pi_{v_3}(\sigma_{v_2=k_2}(R_2))$
$\pi_{v_1}(R_3)$
0 1 2 3
2
0
1
1 2
$\pi_{v_3}(\sigma_{v_1=k_1}(R_3))$

Algorithm 3



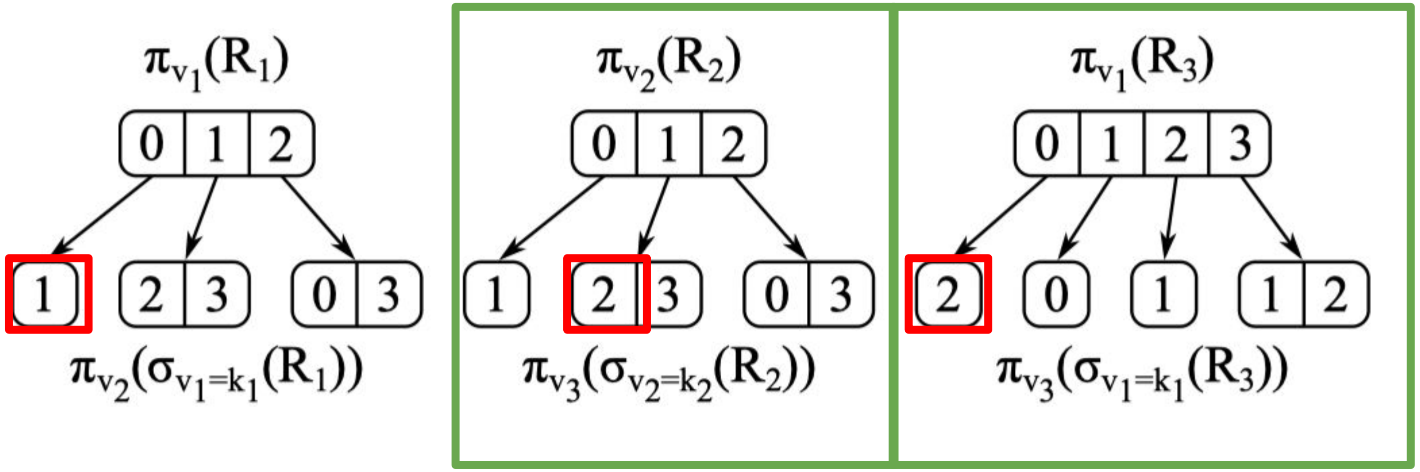
(0,1)

Algorithm 3



(0,1)

Algorithm 3



(0,1) (1,2) (0,2)

Algorithm 3.5

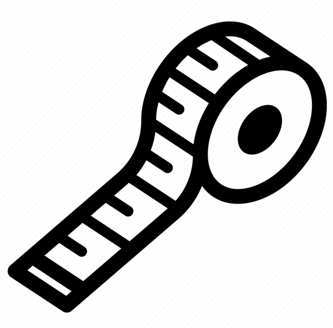
V1 V2 | V2 V3 | V3 V1
(0,1) | (1,2) | (0,2)

(0, 1, 2)

V1 V2 V3

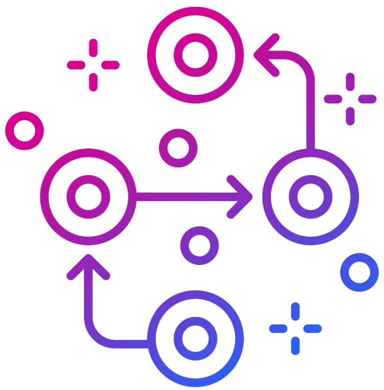
What/How/Where to Measure

- Synthetic integer dataset derived from IMDB dataset
- Count cycles using RDTSC
- Throughput in MB/s
- 11th Gen Intel(R) Core(TM) i7-1165G7 (Tiger Lake) @ 2.80GHz
  L1-I: 32KB, L1-D: 48KB, L2: 1.25MB, L3:12MB
  GCC version: 9.4.0



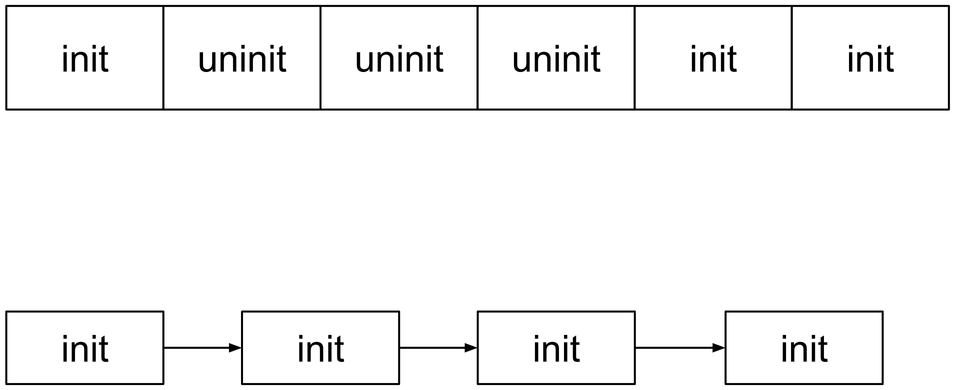
Methodology

- Linux Perf
  - Identify cache misses
  - Page faults
  - Branch misses
- Comment out parts of code



Baseline

- Chain occupied entries in a linkedlist to iterate
- Reduce memory access at cost of losing space locality

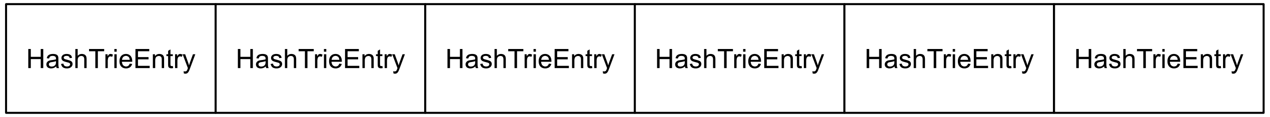


Compiler Flags

```
FLAGS = -march=native -O3
```

Optimization 1 - Optimize Hash Table Format

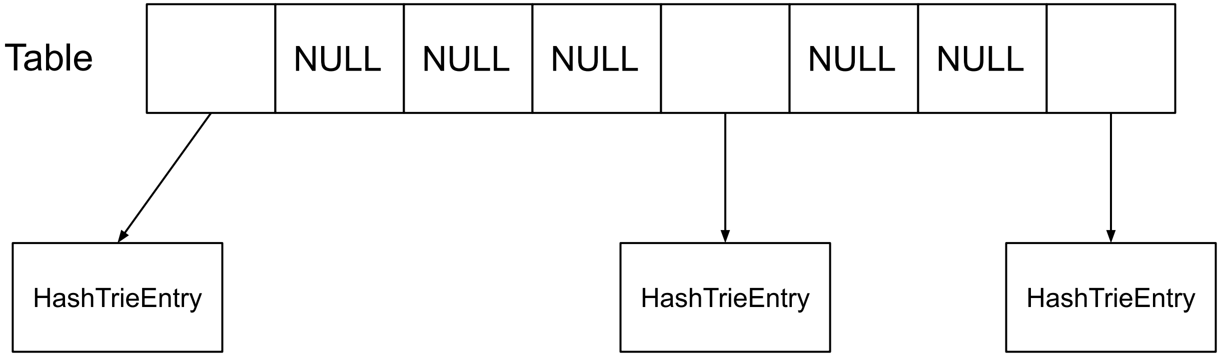
Hash Table




```
hash_table = new HashTrieEntry[allocated_size_arg];
```

```
hash_table = (HashTrieEntry**)calloc(allocated_size_arg, sizeof(HashTrieEntry*));
```

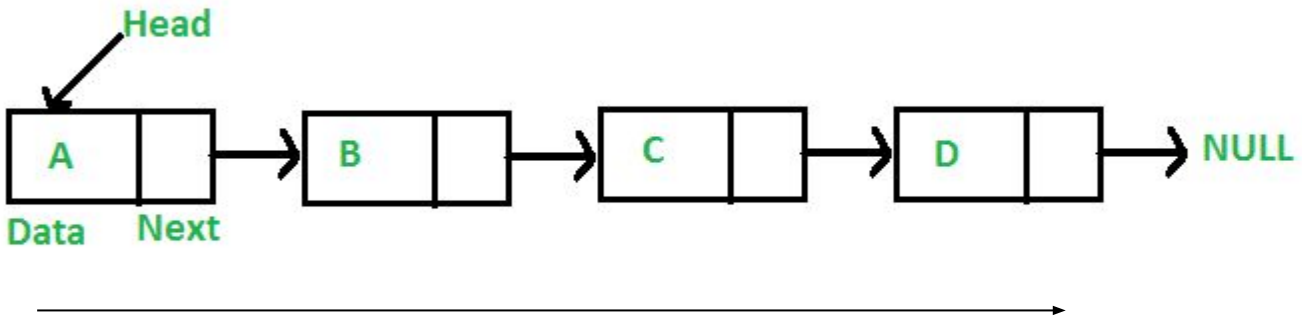
Hash Table




```
HashTrieEntry **hash_table;
```

Optimization 2 - Optimize Tuple List Length Calculation

TupleList::length()




```
int TupleList::length() {
    Tuple *cursor = head;
    int length = 0;
    while(cursor->next) {
        length++;
        cursor = cursor->next;
    }

    return length;
}
```

Optimization 2 - Optimize Tuple List Length Calculation

```cpp
int length() { return len; }
```

```cpp
void TupleList::append(Tuple *node) {
    tail->next = node;
    len++;
    while (tail->next != nullptr) {
        tail = tail->next;
    }
}
```

Optimization 3 - Optimize Tuple Builder

```
/* duplicate all entries in the table n times */
void JoinedTupleBuilder::duplicate(int n) {
    int len = data.size();

    for(int j = 0; j < n; ++j) {
        for(int i = 0; i < len; ++i) {
            data.push_back(data[i]);
        }
    }
}
```

1. Reserve the memory for data to avoid reallocations, the new size is known.

2. Use copy instead of push_back, to avoid unnecessary checks.

```
/* duplicate all entries in the table n times */
void JoinedTupleBuilder::duplicate(int n) {

    // OPTIMIZATION
    int originalSize = data.size();
    int newSize = originalSize * (n + 1);

    // reserve the memory to avoid reallocations
    data.reserve(newSize);

    // copy instead of push_back to avoid unnecessary checks
    for (int i = 1; i <= n; ++i) {
        std::copy(data.begin(), data.begin() + originalSize, std::back_inserter(data));
    }
}
```

Optimization 4 - Work on Algorithm 3.5

V1 V2 V2 V3 V3 V1

(2,3) (3,4) (4,2)

v1	**v2**	**v3**
0	1	2
4	5	2
2	3	4

Optimization 4 - Work on Algorithm 3.5

- Make the result table column-order
- No more deduplication on the fly
- Naively append the rows with duplication

result table

v1	0	4	5	2	← 2
v2	1	5	3	3	← 3
v2	1	5	3	3	← 3
v3	2	2	1	4	← 4
v3	2	2	1	4	← 4
v1	0	4	5	2	← 2

Optimization 4 - Work on Algorithm 3.5

- Make the result table column-order
- No more deduplication on the fly
- Naively append the rows with duplication
- Compact table afterwards

v1	0	4	5	2
v2	1	5	3	3
v2	1	5	3	3
v3	2	2	1	4
v3	2	2	1	4
v1	0	4	5	2

v1	0	4	5	2
v2	1	5	3	3
v3	2	2	1	4

Comparison Between Optimizations 3 and 4

Optimization 3	**Metric**	**Optimization 4**
53’909’382	instructions	41’952’826
11’076’692	branches	8’214’855
37’252	branch-misses	34’693
4’293	LLC-load-misses	3’348
50’878	cache-misses	46’663

Optimization 5 - Optimize Hash Table Lookup

- Remove unnecessary and expensive modulo operation
- Old:

```
do {
        index = (index+1)%cursor->allocated_size;
        [...]
    } while(start != index);
```

- New:

```
for(int i=start; i<allocated_size; ++i) {
      [...]
}
for(int i=0; i<start; ++i) {
       [...]
}
```

Optimization 6 - Optimize Hash Trie Tuple Insertion

- Hash Trie Node, tuple insertion optimization
- Similar to optimization 5
- Old:

```
while(hash_table[index]->hash != hash) {
        index++;
        index %= allocated_size;
    [...]}
```

- New:

```
while(hash_table[index]->hash != hash) {
        index++;
        if(index == allocated_size)
            index=0;
    [...]}
```

Optimization 7 - Optimize Hash Table Format

- Remove hash function
- Move from *uint64_t* to *int*
- More compact data structures
- Better cache locality

Old Hash Table

New Hash Table

Comparison Between Optimizations 6 and 7

Optimization 6	**Metric**	**Optimization 7**
37’193’154	instructions	22’516’258
9’229’755	branches	4’535’679
34’435	branch-misses	30’457
4’132	LLC-load-misses	3’394
46’396	cache-misses	45’087

Optimization 8 - Remove Modulo Operations

```
int start = (index+1) % allocated_size;


for(int i=start; i<allocated_size; ++i) {
    cursor_entry = cursor->hash_table[i];
    if(!cursor_entry){
        return false;
    }
    if(cursor_entry->hash == hash) {
        entry = cursor_entry;
        return true;
    }
}

// if there is an entry at given hash table
// therefore we need to iterate over the ne
for(int i=0; i<start; ++i) {
```

Optimization 8 - Remove Modulo Operations

```
int i = (index+1);

for(; i<allocated_size; ++i) {
    cursor_entry = cursor->hash_table[i];
    if(!cursor_entry){
        return false;
    }
    if(cursor_entry->hash == hash) {
        entry = cursor_entry;
        return true;
    }
}

// if there is an entry at given hash tabl
// therefore we need to iterate over the n
for(i=0; i<=index; ++i) {
    cursor_entry = cursor->hash_table[i];
```

Optimization 9 - Vector Intrinsics

```
int col_index_in_builder_columns = table_start_indices[table_index] + col_index;
for (int row_index = 0; row_index < num_rows; row_index++) {
    col_table->data[current_col_index][row_index] = columns[col_index_in_builder_columns][row_index];
}
current_col_index++;
```

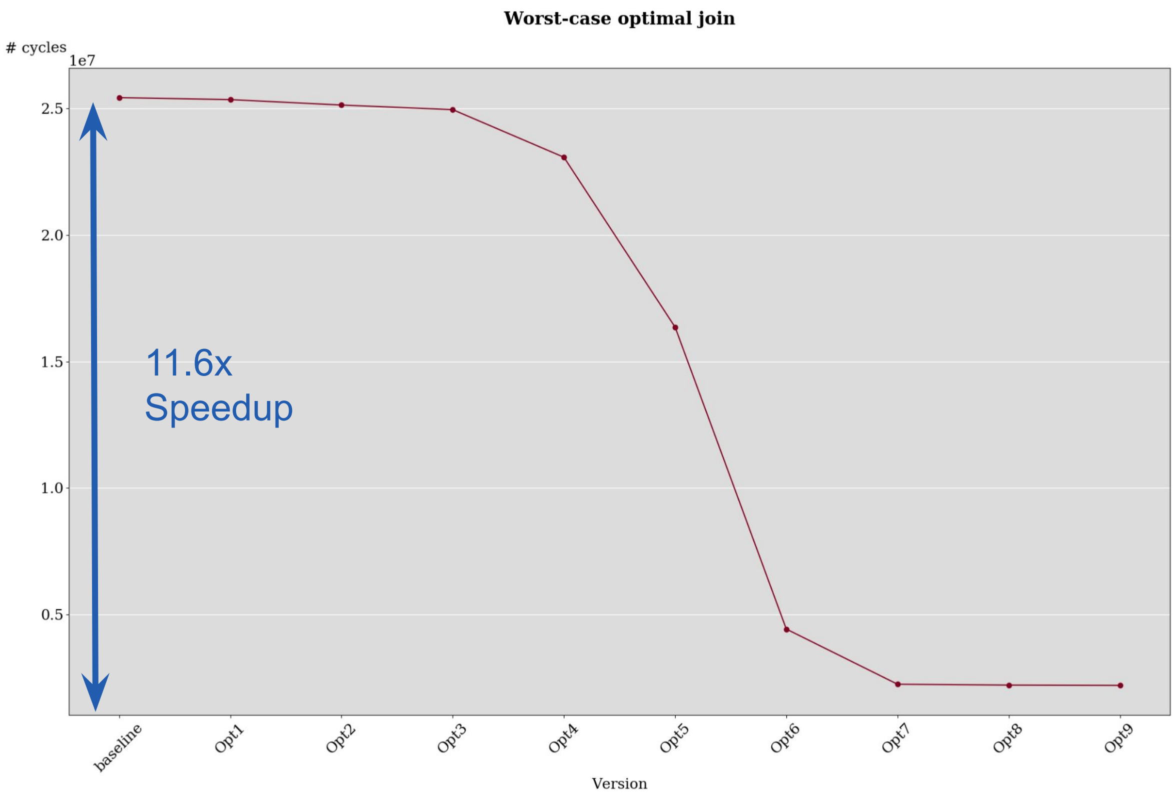
Optimization 9 - Vector Intrinsics

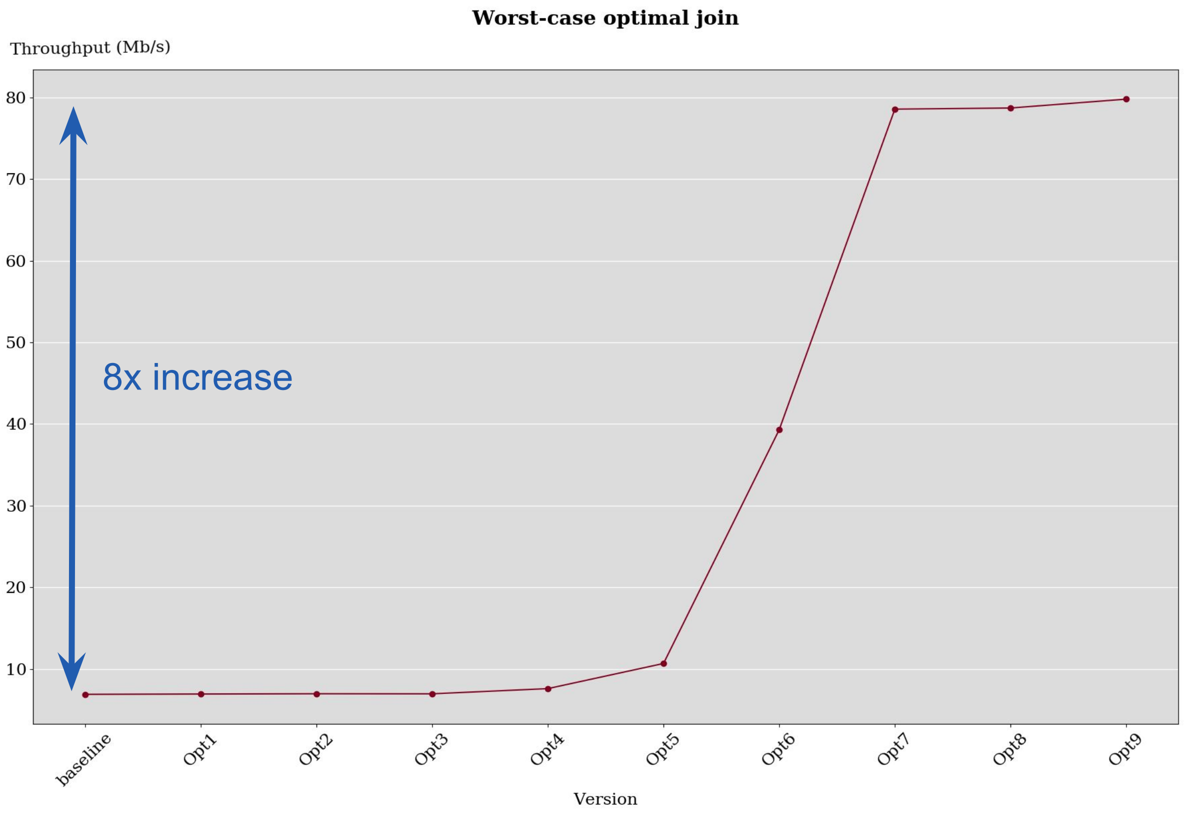
```
int col_index_in_builder_columns = table_start_indices[table_index] + col_index;
int* col = columns[col_index_in_builder_columns].data();
int row_index = 0;
int *base = col_table->data[current_col_index];

for (; row_index < num_rows - 7; row_index+=8) {
    __m256i t = _mm256_loadu_si256((__m256i*)(col + row_index));
    _mm256_storeu_si256((__m256i*)(base + row_index), t);
}

for (; row_index < num_rows; row_index++) {
    col_table->data[current_col_index][row_index] = col[row_index];
}
current_col_index++;
```

Comparison Between Baseline and Last Version

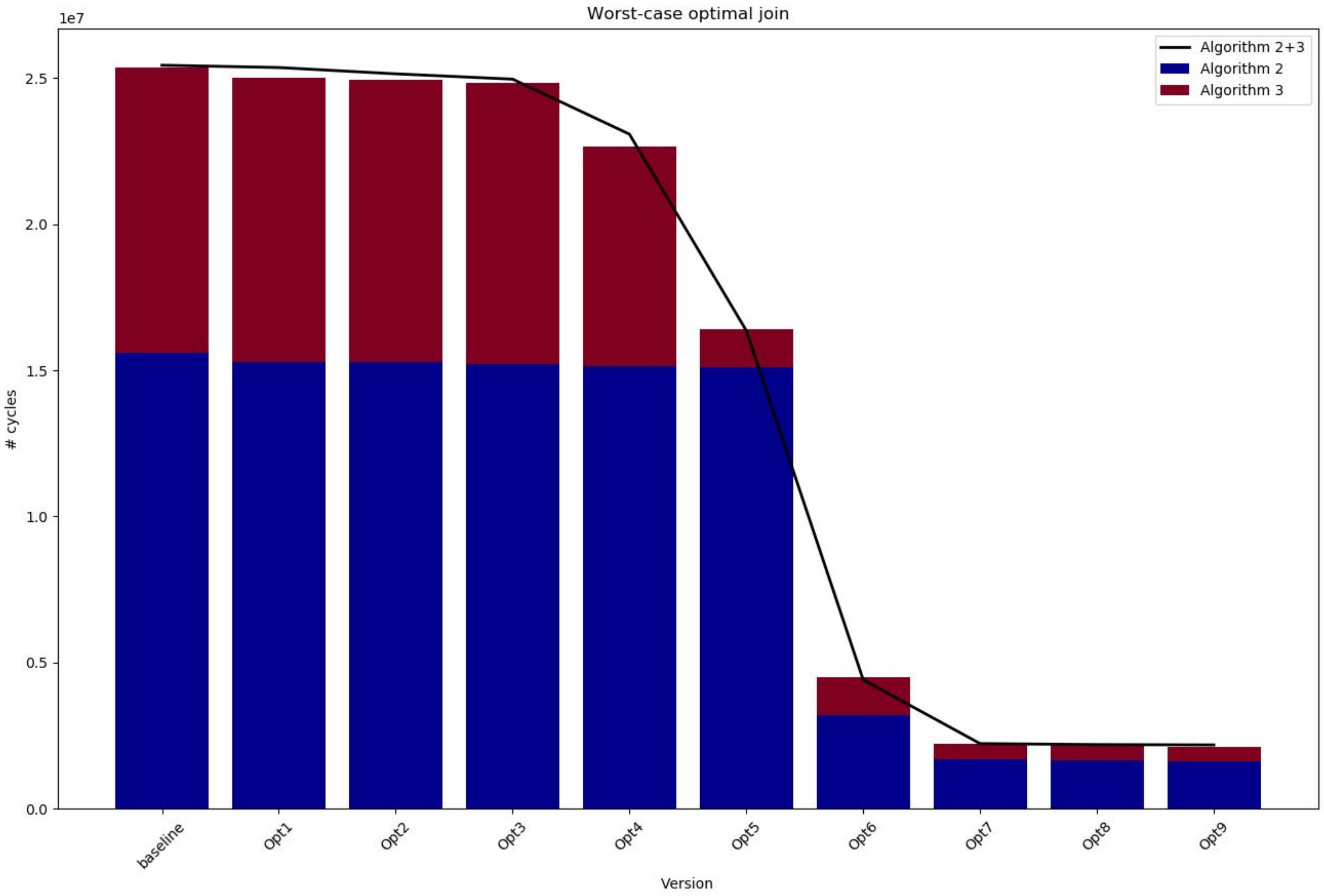
Baseline	Metric	Last Version
56’438’587	instructions	22’410’126
11’358’037	branches	4’525’491
39’773	branch-misses	30’509
4’892	LLC-load-misses	3’075
46’079	cache-misses	43’178

Performance Results





Performance Results



ETH zürich

Results FOR LARGE DATASET

Conclusion

- Heavily memory bound
- Not optimal spatial locality in hash trie
- Recursive algorithm
- Still achieved 11.6x speedup