Task

- Output results of SQL queries with joins as fast as possible
- At the start of the program, relations are loaded from binary files
- There is 1 second at the beginning available for preprocessing
- The program then receives SQL queries in batches

Input

Output

```
SELECT SUM(R1.a), SUM(R2.b)
FROM R1
JOIN R2 ON R1.c = R2.d
WHERE R2.a > 5 AND R1.b = 8
```

```
1865, 1337
```

Query processing overview

1. Rewrite the query
   - Quickly filter out empty queries and redundant joins
2. Create indices for all joined columns
   - Clustered indices: full copies of relations sorted by the joined column
   - The workload is read only: indices are fully cached in-memory
3. Build an operator tree from the query
   - Create a left-deep join tree
   - Use merge-sort join and nested loop join with index accesses
   - Operators are fully pipelined (tuple-at-a-time)
4. Split the operator tree into hundreds of disjoint tasks
5. Execute the tasks in parallel and output the results

Workload statistics

- Read-only: everything can be cached
- Average column count: 3
- Large variation of row count: from 8k to 20 million
- Most columns are uniformly distributed
- ~40% of queries can be skipped after rewriting

Operator tree

- Merge-sort is used if possible (both columns sorted)
- Nested loop with index access is used otherwise
- Many virtual calls: mark leaves with `final`, inline hot methods
- Standard rules applied: projection and selection pushdown
- The plan is split into many tasks and executed in parallel

Query rewriting

- Remove redundant projections and selections
- Remove joins of foreign-primary key column pairs
- Find bounds of join components and add them as new selections
- Use indices to lookup join ranges: if empty, the query can be skipped
- ~40% of queries eliminated, 40% of queries skipped

```
SELECT SUM(R1.a)
FROM R1
JOIN R2 ON R1.c = R2.d
JOIN R3 ON R1.d = R3.a
WHERE R2.a > 5
```

```
SUM(R1.a)
```

```
NJ
```

```
MSJ
Index
```

```
R1 Filter
```

```
R3
```

Indices

- Clustered indices: full copies of relations sorted by a specific column
- Lookup: binary search, O(log n)
- Index build initially very slow because of slow sorting

Index build strategy

1. Find min/max of the sorted column
2. Divide column values into 512 KiB groups (parallel)*
3. Sort groups with MSB radix sort (parallel)
* division into groups assumes uniform distribution

Inner aggregation

- Summing with tuple-by-tuple access has high overhead
- Leverage repeating values in sorted columns
- Directly aggregate columns and return (sum, row count)
- Final value: left sum * right count, right sum * left count
- ~2x faster query processing

```
SELECT SUM(R1.b), SUM(R2.b)
FROM R1
JOIN R2 ON R1.a = R2.a
WHERE R1.b > 5 AND R2.a > 8
```

```
MSJ
Index
```

```
R1 Filter
```

```
R2
```

```
R3
```

General optimizations

- Spread memory amongst nodes with first-touch policy
- Spread OpenMP threads amongst sockets (OMP_PLACES=sockets)
- Near NUMA memory accesses ~2x faster than far accesses

JIT compile filters (selections)

1. `map` a block of executable memory
2. Compile filter predicates to x64 instructions
3. Run filters directly as functions without interpretation
- Can be done with ~50 lines of C++
- Use godbolt.org for instruction opcodes

Force vectorization

- Move vectorizable loops to separate functions
- Mark pointers with `restrict` and `const` if possible
- Keep the function simple to make it easy for the compiler

Fun facts

- ~7200 lines of C++ 14
- 1625 submits
- Index build takes ~50% of execution time

Third-party libraries

- OpenMP
- radix sort (https://github.com/voutcn/kxsort)