CWI

Database Architectures

DuckDB

An Embeddable Analytical Database

Mark & Hannes

Agenda

- **Motivation**
- Design
- Implementation
- Testing
- Next Steps

DB is missing the boat again

- More and more complex Python/R/Julia/… libraries being deployed to solve basic relational problems
- DB world largely irrelevant there. Why?
    - Embeddability
    - Ease of Use

What about SQLite ?

- In-process SQL database, data either in memory or in a file, rock-solid, used on every smartphone, browser, OS, ….
- People also use it for large-ish dataset analysis
- Bad idea, SQLite was never built for this
  - e.g. row-based storage model

What about MonetDBLite?

- Attempted to re-design existing system. Sort of worked.
- Problems:
  - Error handling, global variables, restart, multi-DB, …
  - Memory management & resource allocation difficult
  - Problematic processing paradigm for embedded
  - Bulk intermediates require lots of memory and/or disk space and interfere with host
  - No graceful handling of out-of-memory situations

	OLTP	OLAP
Embedded	SQLite	?
Stand-Alone	PostgreSQL IBM DB2	TERADATA ClickHouse

Agenda

- Motivation
- **Design**
- Implementation
- Testing

DuckDB Goals

- Fast OLAP, reasonable OLTP
  - e.g. concurrent appends
- Fully and easily embeddable
  - No globals, no dependencies
  - Works gracefully in low or out-of-memory situations
- Stability (aspiring to match SQLite)
- Clean, readable, consistent and extensible code
  - Basis for future research projects
- Full-featured and in public use

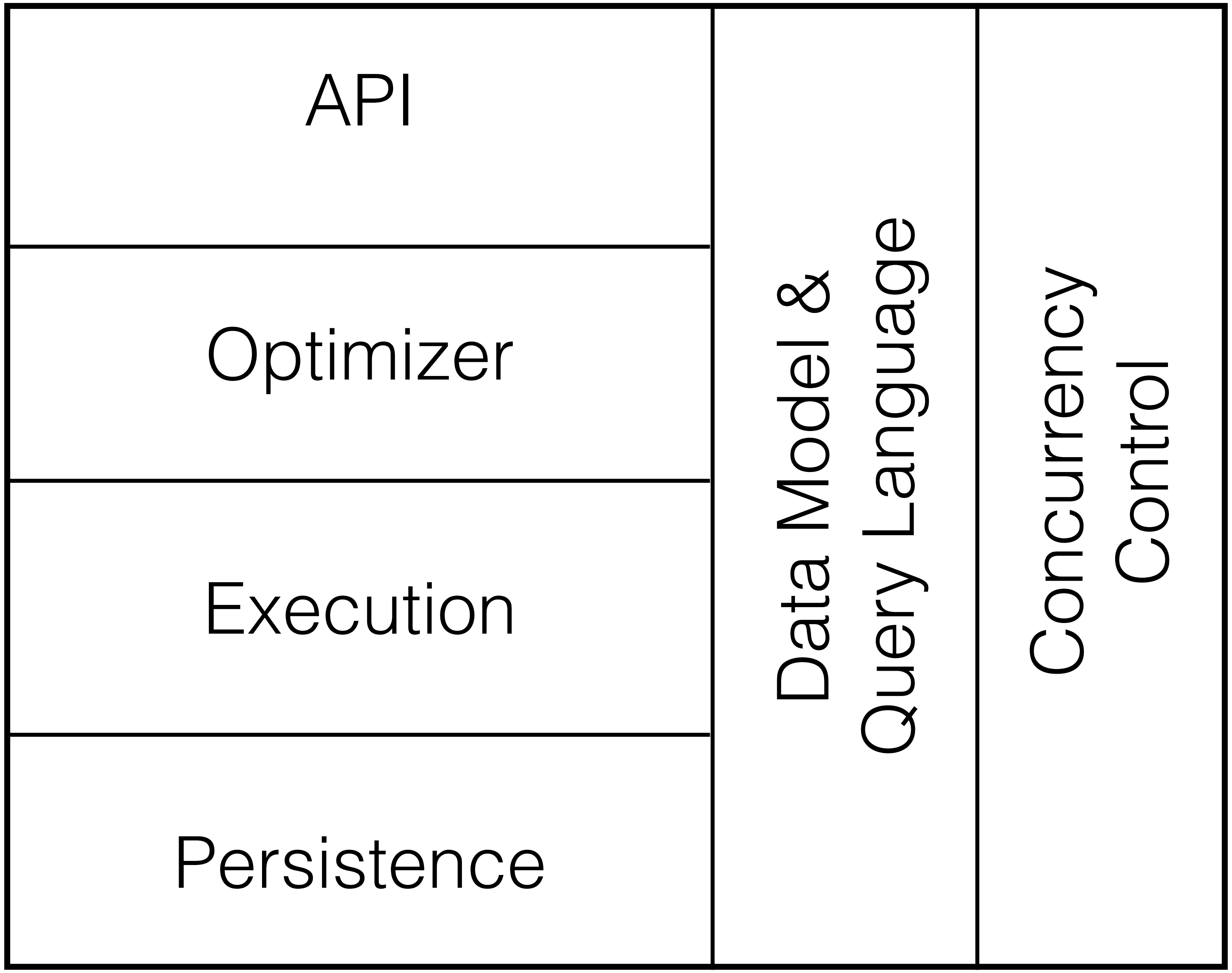
DuckDB Design Choices

- SQL and relations
- Vectorized Model
    - because JIT has too many dependencies
- MVCC
    - e.g. concurrent appends

Agenda

- Motivation
- Design
- **Implementation**
- Testing
- Next Steps

Architecture Overview



API

Optimizer

Execution

Persistence

Data Model &
Query Language

Concurrency
Control

Implementation

- **C++ 11**
- Rich & efficient and stable standard library
  - no need to roll own lists/hash
  - not used for actual data but for all auxiliary structures
- Scopes do memory/lock management for us
  - unique_ptrs, avoid leaks
  - lock_guard, auto-acquiring and releasing locks
  - destructors
- OOP, information hiding, clean namespaces
  - important for embedding

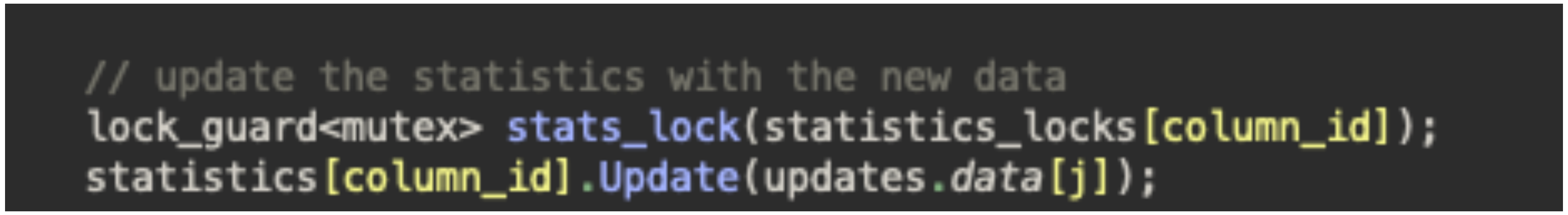
C++ STL?

- But “STL is slower than specialized solutions”
    - This is true.
- Specialized solutions for everything are prime example of premature optimization
- Without STL you need to do this for everything
- Often results in slower and buggier code
    - STL is extremely well tested (used by millions)

Exceptions

- Exceptions + smart pointers = automatic cleanup
  - No leaks!
- Exceptions are zero-cost when not triggered
- In DuckDB: Never more than one per query. Ex:
  - Handle OOM
  - Query cancellation
  - Fatal query errors

Exceptions



```
// update the statistics with the new data
lock_guard<mutex> stats_lock(statistics_locks[column_id]);
statistics[column_id].Update(updates.data[j]);
```

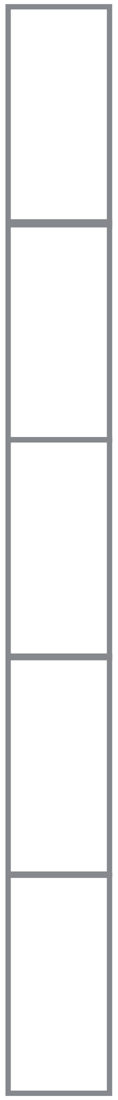
- Statistics::Update can throw an exception
- The lock will be released regardless, without us doing any cleanup

Templates

- Code expansion in C: scripts/macros
- Code expansion in C++: Templates
    - Debuggable, more readable, less error-prone

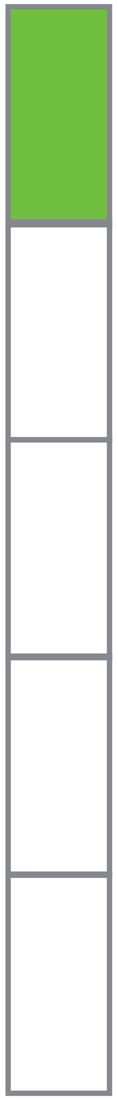
Life of a query

-
  ```
  SELECT count(*)
  FROM lineitem JOIN orders ON
  l_orderkey=o_orderkey
  WHERE o_orderstatus='X' AND l_tax > 50;
  ```



Step 1: Parser

- PostgreSQL parser
  - Battle-tested
  - Side-effect: Postgres compatibility
  - `libpg_query`
- Transform into custom class structure
  - Inspired by Peloton



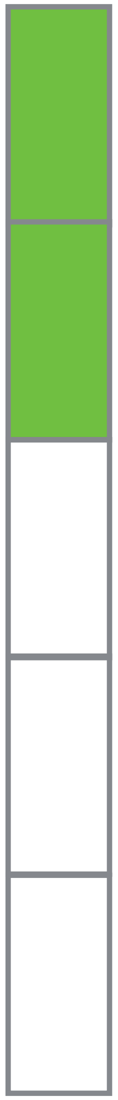
```
//! SelectStatement is a typical SELECT clause
class SelectStatement : public SQLStatement {
  public:
   SelectStatement()
       : SQLStatement(StatementType::SELECT), select_distinct(false),
         union_select(nullptr){};

   //! The projection list
   vector<unique_ptr<Expression>> select_list;
   //! The FROM clause
   unique_ptr<TableRef> from_table;
   //! The WHERE clause
   unique_ptr<Expression> where_clause;
   //! DISTINCT or not
   bool select_distinct;
   //! Group By Description
   GroupByDescription groupby;
   //! Order By Description
   OrderByDescription orderby;
   //! Limit Description
   LimitDescription limit;

   unique_ptr<SelectStatement> union_select;
   unique_ptr<SelectStatement> except_select;
}
```

Step 2: Binder & Planner

- Binder
  - Resolve table and column names
  - Resolve data types
  - Overflow prevention!
    - Data statistics used for type promotion if required
    - Statistics right now: min, max, max str length
- Planner
  - Transform parse tree into logical operator tree



```
SELECT count(*)
FROM lineitem JOIN orders ON l_orderkey=o_orderkey
WHERE o_orderstatus='X' AND l_tax > 50;
```

```
AGGREGATE_AND_GROUP_BY[COUNT_STAR]
 FILTER[(l_tax > CAST[DECIMAL](50)), (o_orderstatus = X)]
  JOIN[EQUAL(l_orderkey, o_orderkey)]
   GET(lineitem)
   GET(orders)
```

Step 3: Optimizer

- Rule-based optimizer
    - Matches tree patterns
- No join ordering yet

```
ConstantFoldingRule::ConstantFoldingRule() {
    root = std::unique_ptr<AbstractRuleNode>(new ExpressionNodeSet(
        {ExpressionType::OPERATOR_ADD, ExpressionType::OPERATOR_SUBTRACT,
         ExpressionType::OPERATOR_MULTIPLY, ExpressionType::OPERATOR_DIVIDE,
         ExpressionType::OPERATOR_MOD}));
    root->children.push_back(
        make_unique_base<AbstractRuleNode, ExpressionNodeType>(
            ExpressionType::VALUE_CONSTANT));
    root->children.push_back(
        make_unique_base<AbstractRuleNode, ExpressionNodeAny>());
    root->child_policy = ChildPolicy::UNORDERED;
}
```

```
AGGREGATE_AND_GROUP_BY[COUNT_STAR]
 FILTER[(l_tax > CAST[DECIMAL](50)), (o_orderstatus = X)]
  JOIN[EQUAL(l_orderkey, o_orderkey)]
   GET(lineitem)
   GET(orders)
```

↓

```
AGGREGATE_AND_GROUP_BY[COUNT_STAR]
 JOIN[EQUAL(l_orderkey, o_orderkey)]
  FILTER[(l_tax > 50.000000)]
   GET(lineitem)
  FILTER[(o_orderstatus = X)]
   GET(orders)
```

Pushdown!

Step 4: Physical Planner

- Selects physical implementation for logical operators

```
void PhysicalPlanGenerator::Visit(LogicalJoin &op) {
    if (has_equality) {
        // equality join: use hash join
        plan = make_unique<PhysicalHashJoin>(move(left), move(right),
                                             move(op.conditions), op.type);
    } else {
        // non-equality join: use nested loop
        if (op.type == JoinType::INNER) {
            plan = make_unique<PhysicalNestedLoopJoinInner>(
                move(left), move(right), move(op.conditions), op.type);
        } else if (op.type == JoinType::ANTI || op.type == JoinType::SEMI) {
            plan = make_unique<PhysicalNestedLoopJoinSemi>(
                move(left), move(right), move(op.conditions), op.type);
        } else {
            throw NotImplementedException(
                "Unimplemented nested loop join type!");
        }
    }
}
```

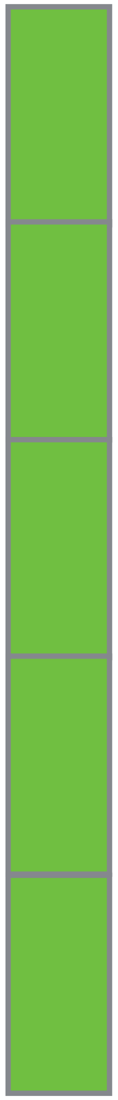
```
AGGREGATE_AND_GROUP_BY[COUNT_STAR]
 JOIN[EQUAL(l_orderkey, o_orderkey)]
  FILTER[(l_tax > 50.000000)]
   GET(lineitem)
  FILTER[(o_orderstatus = X)]
   GET(orders)
```

↓

```
HASH_GROUP_BY[COUNT_STAR]
 HASH_JOIN[EQUAL(l_orderkey, o_orderkey)]
  FILTER[(l_tax > 50.000000)]
   SEQ_SCAN[lineitem]
  FILTER[(o_orderstatus = X)]
   SEQ_SCAN[orders]
```

Step 5: Execution

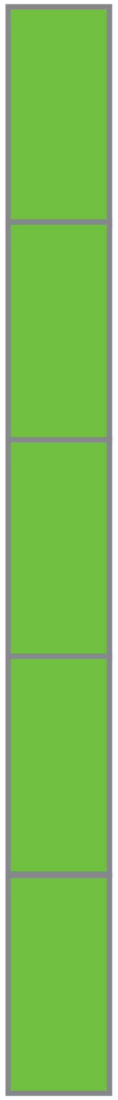
- **DataChunk** with max. length 1024 (Table slice)
    - **Vectors**, which are native arrays of certain type (int, float etc.)
    - NULL masks (16 x 8 byte integers per vector!)
        - Can be inherited or OR-ed together for vector operations
    - Selection vectors
- “Vector-Volcano”: pull DataChunk from root node of plan
    - Continue until result is empty, query then finished
    - Early materialisation
- Physical operators implemented using library of vector operations



Query Profiling

```
--------------------
|   HASH_GROUP_BY  |
|      (0.00s)     |
|         1        |
--------------------
--------------------
|     HASH_JOIN    |
|       INNER      |
|    l_orderkey    |
|    =o_orderkey   |
|      (0.00s)     |
|         0        |
--------------------
----------------------------------------
|      FILTER      ||      FILTER      |
|     l_tax >      ||  o_orderstatus   |
|     50.000000    ||       = X        |
|      (0.00s)     ||     (0.00s)      |
|         0        ||        0         |
----------------------------------------
----------------------------------------
|     SEQ_SCAN     ||     SEQ_SCAN     |
|     lineitem     ||      orders      |
|      (0.00s)     ||     (0.00s)      |
|         0        ||     150000       |
----------------------------------------
```

Never scanned
because RHS empty



DataChunk & Vector

```
class Vector {
  public:
   //! The type of the elements stored in the vector.
   TypeId type;
   //! The amount of elements in the vector.
   size_t count;
   //! A pointer to the data.
   char *data;
   //! The selection vector of the vector.
   sel_t *sel_vector;
   //! The null mask of the vector, if the Vector has any NULL values
   nullmask_t nullmask;
}

class DataChunk {
  public:
   //! The amount of vectors that are part of this DataChunk.
   size_t column_count;
   //! The vectors owned by the DataChunk.
   std::unique_ptr<Vector[]> data;
   //! The (optional) selection vector of the DataChunk. Each of the member
   //! vectors reference this selection vector.
   sel_t *sel_vector;
}
```

VectorOperations

```
void VectorOperations::Add(Vector &left, Vector &right, Vector &result) {
   switch (left.type) {
   …
   case TypeId::INTEGER:
      templated_binary_loop<int32_t, operators::Add>(
          left, right, result);
      break;
   …
}
```

```
struct Add {
   template <class T> static inline T Operation(T left, T right) {
      return left + right;
   }
};
```

Template Magic

```
template <class T, class OP>
void templated_binary_loop(Vector &left, Vector &right, Vector &result) {
    auto ldata = (T *)left.data;
    auto rdata = (T *)right.data;
    auto result_data = (T *)result.data;

    result.nullmask = left.nullmask | right.nullmask;
    binary_loop_function_array<T, OP>(
        ldata, rdata, result_data, left.count, left.sel_vector);

    result.sel_vector = left.sel_vector;
    result.count = left.count;
}
```

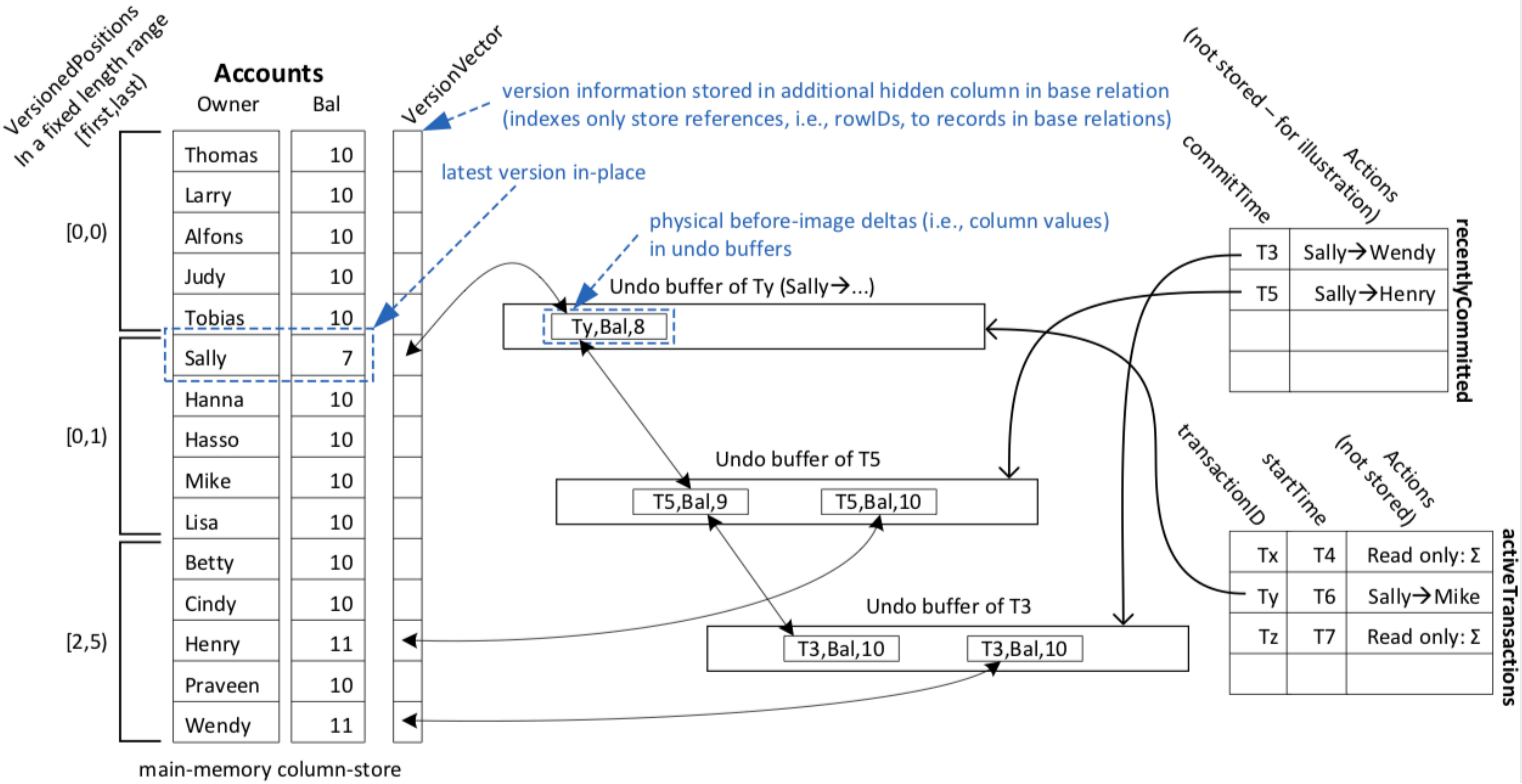
```
template <class T, class OP>
static inline void
binary_loop_function_array(T *__restrict ldata,
                                                  T *__restrict rdata,
                                                  T *__restrict result_data, size_t count,
                                                  sel_t *__restrict sel_vector) {
    if (sel_vector) {
        for (size_t i=0; i < count; i++) {
          result_data[sel_vector[i]] = OP::Operation(ldata[sel_vector[i]], rdata[sel_vector[i]]);
        }
    } else {
        for (size_t i=0; i < count; i++) {
          result_data[i] = OP::Operation(ldata[i], rdata[i]);
        }
    }
}
```

Templating result:

```
for (size_t i = 0; i < count; i++) {
 result_data[i] = ldata[i] + rdata[i];
}
```

- Tight loops for every type
- Compiler will SIMDize loops
    - restrict in template

MVCC Design



Accounts

Owner	Bal
Thomas	10
Larry	10
Alfons	10
Judy	10
Tobias	10
Sally	7
Hanna	10
Hasso	10
Mike	10
Lisa	10
Betty	10
Cindy	10
Henry	11
Praveen	10
Wendy	11

version information stored in additional hidden column in base relation (indexes only store references, i.e., rowIDs, to records in base relations)

latest version in-place

physical before-image deltas (i.e., column values) in undo buffers

recentlyCommitted

commitTime	Actions (not stored – for illustration)
T3	Sally→Wendy
T5	Sally→Henry

activeTransactions

transactionID	startTime	Actions (not stored)
Tx	T4	Read only: Σ
Ty	T6	Sally→Mike
Tz	T7	Read only: Σ

"Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems"
Thomas Neumann, Tobias Mühlbauer and Alfons Kemper.

DuckDB API

- Main API: C++

```
DuckDB db(nullptr);
DuckDBConnection con(db);

auto res = con.Query("SELECT 42");
int result = res->GetValue<int>(0, 0);
```

- C API
  - `duckdb_open()`
  - `duckdb_connect()`
  - `duckdb_query()`
  - …
- SQLite API wrapper (same header)
- SQLite shell (demo!)

Agenda

- Motivation
- Design
- Implementation
- **Testing**
- Next Steps

Testing overview

- Unit tests with Catch2
  - core: ~one minute (demo)
  - extended: 30 mins
- SQLite `sqllogictests`
- `sqlsmith`
- Continuous integration with Jenkins
- Continuous code coverage with lcov

Catch2

build passing coverage 66.11%

Example Unit Test Case

```
#include "catch.hpp"

TEST_CASE("Test LEFT OUTER JOIN", "[join]") {
   unique_ptr<DuckDBResult> result;
   DuckDB db(nullptr);
   DuckDBConnection con(db);

   con.Query("CREATE TABLE integers(i INTEGER, j INTEGER)”);
   con.Query("INSERT INTO integers VALUES (1, 2), (2, 3), (3, 4)")
   con.Query("CREATE TABLE integers2(k INTEGER, l INTEGER)");
   con.Query("INSERT INTO integers2 VALUES (1, 10), (2, 20)");

   result = con.Query("SELECT * FROM integers LEFT OUTER JOIN integers2 ON "
                      "integers.i=integers2.k ORDER BY i”);

   REQUIRE(CHECK_COLUMN(result, 0, {1, 2, 3}));
   REQUIRE(CHECK_COLUMN(result, 1, {2, 3, 4}));
   REQUIRE(CHECK_COLUMN(result, 2, {1, 2, Value()}));
   REQUIRE(CHECK_COLUMN(result, 3, {10, 20, Value()}));
}
```

Continuous Benchmarking

- Result verification and performance testing
- First correct, then fast
- Benchmarks
  - Microbenchmarks
  - TPC-H (complete)
  - TPC-DS (73/103 queries run)
  - TPC-E (only data generation)

```
DuckDB db(nullptr);
DuckDBConnection con(db);
tpch::dbgen(1, db);
```

- All use *in-process data generation*

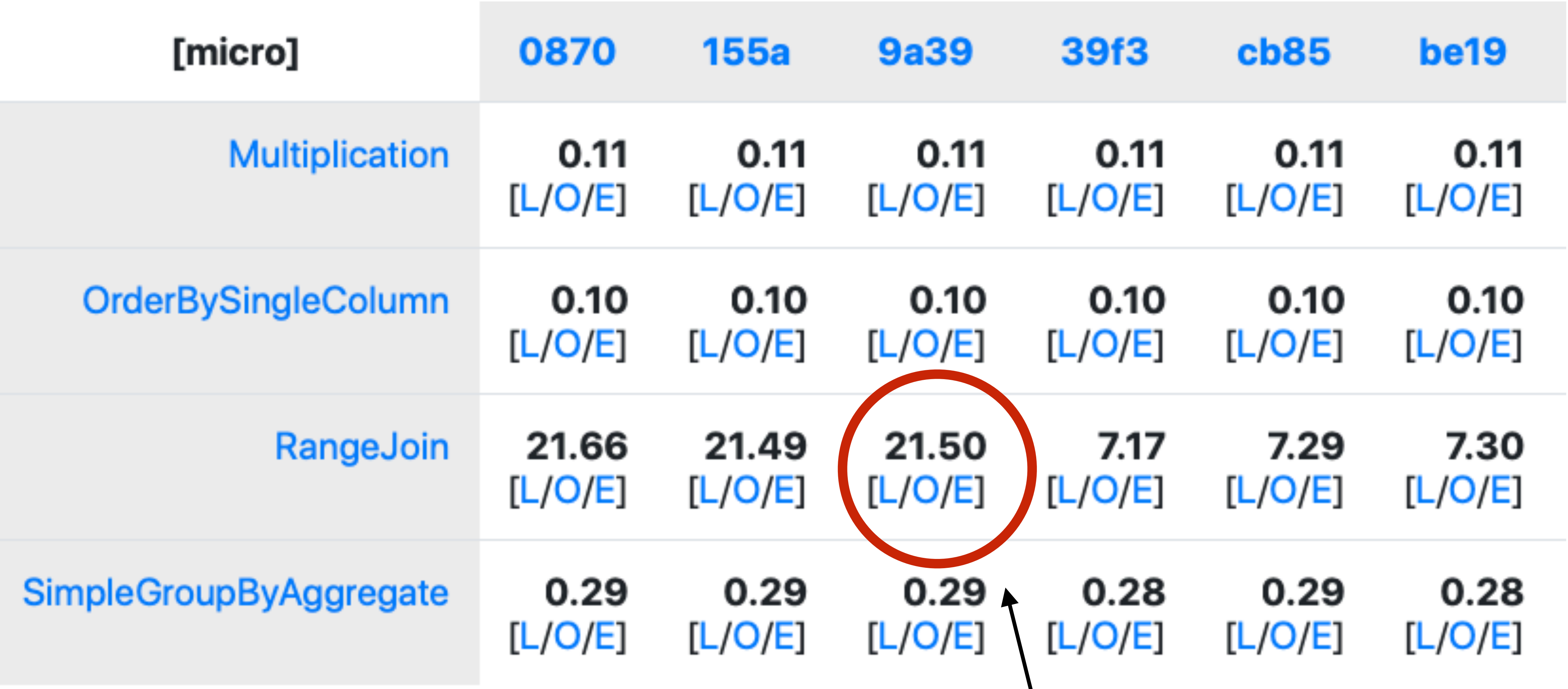
```cpp
DUCKDB_BENCHMARK(RangeJoin, "[micro]")
virtual void Load(DuckDBBenchmarkState *state) {
	// fixed seed random numbers
	std::uniform_int_distribution<> distribution(1, 10000);
	std::mt19937 gen;
	gen.seed(42);

	state->conn.Query("CREATE TABLE integers(i INTEGER, j INTEGER);");
	auto appender = state->conn.GetAppender("integers");
	// insert the elements into the database
	for (size_t i = 0; i < RANGEJOIN_COUNT; i++) {
		appender->begin_append_row();
		appender->append_int(distribution(gen));
		appender->append_int(distribution(gen));
		appender->end_append_row();
	}
	state->conn.DestroyAppender();
}

virtual std::string GetQuery() {
	return "SELECT * FROM integers a, integers b WHERE (a.i / 1000) > b.j;";
}

virtual std::string VerifyResult(DuckDBResult *result) {
	if (!result->GetSuccess()) {
		return result->GetErrorMessage();
	}
	return std::string();
}
FINISH_BENCHMARK(RangeJoin)
```

Git revisions



[micro]	0870	155a	9a39	39f3	cb85	be19
Multiplication	0.11 [L/O/E]	0.11 [L/O/E]	0.11 [L/O/E]	0.11 [L/O/E]	0.11 [L/O/E]	0.11 [L/O/E]
OrderBySingleColumn	0.10 [L/O/E]	0.10 [L/O/E]	0.10 [L/O/E]	0.10 [L/O/E]	0.10 [L/O/E]	0.10 [L/O/E]
RangeJoin	21.66 [L/O/E]	21.49 [L/O/E]	21.50 [L/O/E]	7.17 [L/O/E]	7.29 [L/O/E]	7.30 [L/O/E]
SimpleGroupByAggregate	0.29 [L/O/E]	0.29 [L/O/E]	0.29 [L/O/E]	0.28 [L/O/E]	0.29 [L/O/E]	0.28 [L/O/E]

Catches regressions

http://www.duckdb.org/benchmarking/

Agenda

- Motivation
- Design
- Implementation
- Testing
- **Next Steps**

Next Steps

- Help is appreciated, ask us for repo access and send a PR
  - Physical storage and buffer manager (Only WAL at the moment)
  - Foreign keys
  - Join ordering (Idea: use sampling) and more optimiser rules
  - Prepared statements & query cache
  - More types (real decimal, blob, timestamp)
  - More SQL features (`PARTITION` etc.)
  - Intra-query parallelism