Hasso Plattner Institut

IT Systems Engineering | Universität Potsdam

Build your own Database

Week 4

Agenda

- Q&A Sprint 2
- Review Sprint 1
- Experiments with Chunks
- std::optional
- Lambdas

Sprint 2

Questions?

Formatting / Linting

```
 dyod_sprint git:(sprint1_group) gs
On branch: sprint1_group    |   No changes (working directory clean)
 dyod_sprint git:(sprint1_group) ./scripts/format.sh
 dyod_sprint git:(sprint1_group) ✗ gs
On branch: sprint1_group    |   [*] => $e*

Changes not staged for commit

      modified:   [1] src/test/storage/chunk_test.cpp

 dyod_sprint git:(sprint1_group) ✗
```

```
[→  dyod_sprint git:(sprint1_group) ✗ ./scripts/lint.sh
src/test/storage/chunk_test.cpp:29:  Add #include <memory> f
Category 'build/include_what_you_use' errors found: 1
Total errors found: 1
src/test/storage/storage_manager_test.cpp:19:  Add #include
] [4]
Category 'build/include_what_you_use' errors found: 1
Total errors found: 1
→  dyod_sprint git:(sprint1_group) ✗
```

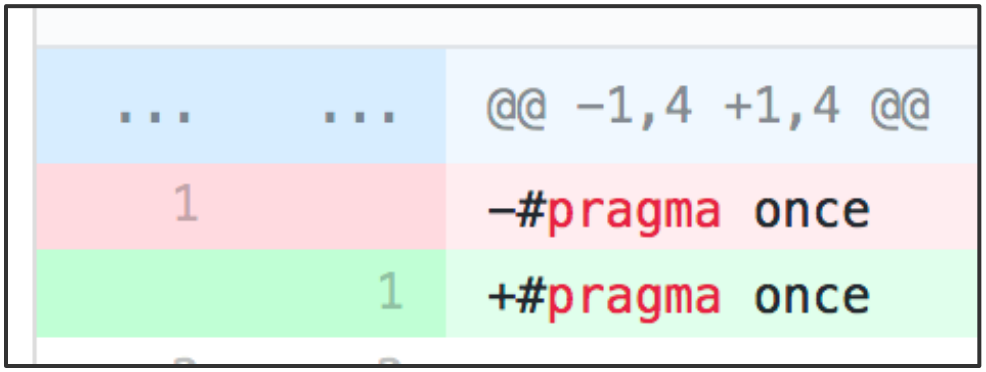
Code Style

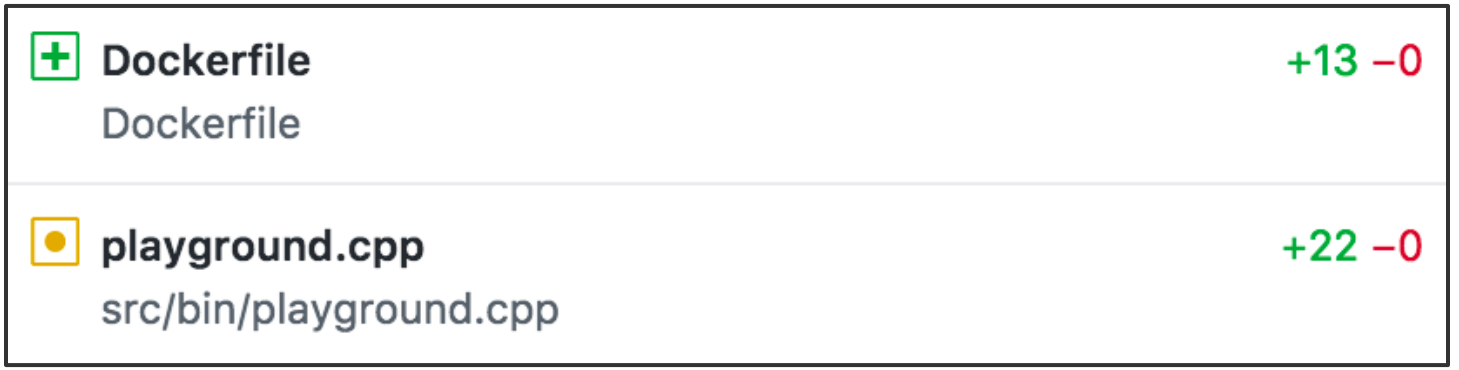
```
std::vector<std::string> m_columnNames;
std::vector<std::string> m_columnTypes;
std::vector<Chunk> m_chunks;
unsigned int m_chunkSize;
```

```
pbpaste | cat | highlight -O rtf --src-lang c++ | pbcopy
```

```
this->chunk_size()
```

Clean Commits





Conceptual Things

```
uint64_t Table::row_count() const {
  return (_table_chunks.size() - 1) * this->chunk_size() +
         _table_chunks.back().size();
}
```

For now, this is not incorrect, but in the future, it will not work any more.

Conceptual Things

```
std::vector<std::string> StorageManager::table_names() const {
  std::vector<std::string> names;
  auto get_name = [](const auto& entry) { return entry.first; };
  std::transform(m_tables.begin(), m_tables.end(), std::back_inserter(names), get_name);
  return names;
}
```

269 characters, lambdas, std::transform, std::back_inserter

```
std::vector<std::string> StorageManager::table_names() const {
  std::vector<std::string> names;
  for (const auto& table_item : _tables) {
    table_names.emplace_back(table_item.first);
  }
  return names;
}
```

209 characters

C++ things

Let's play a different game – what did we *like* about this?

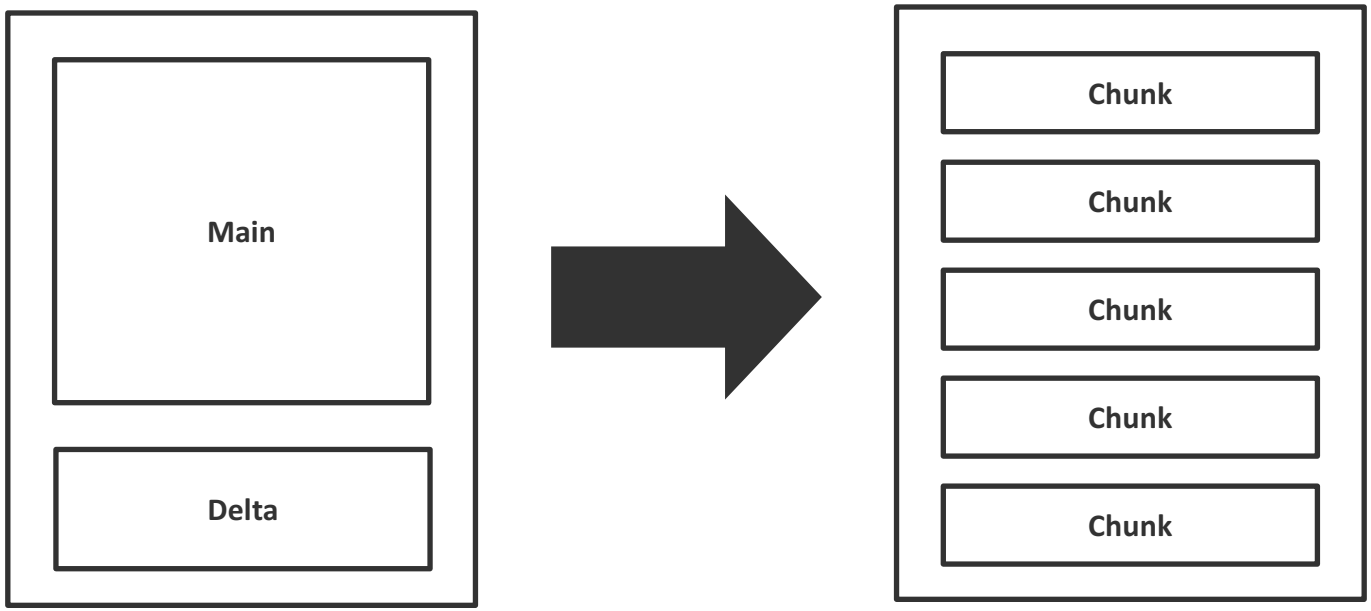
```
std::vector<std::string> StorageManager::table_names() const {
  std::vector<std::string> names;
  names.reserve(m_tables.size());
  // […]
```

```
for (const auto& chunk : m_chunks) {
  count += chunk.size();
}
```

Const


const
const
const
const
const
const
const
const
const
const
const
const

Experiments with Chunks



Experiments with Chunks

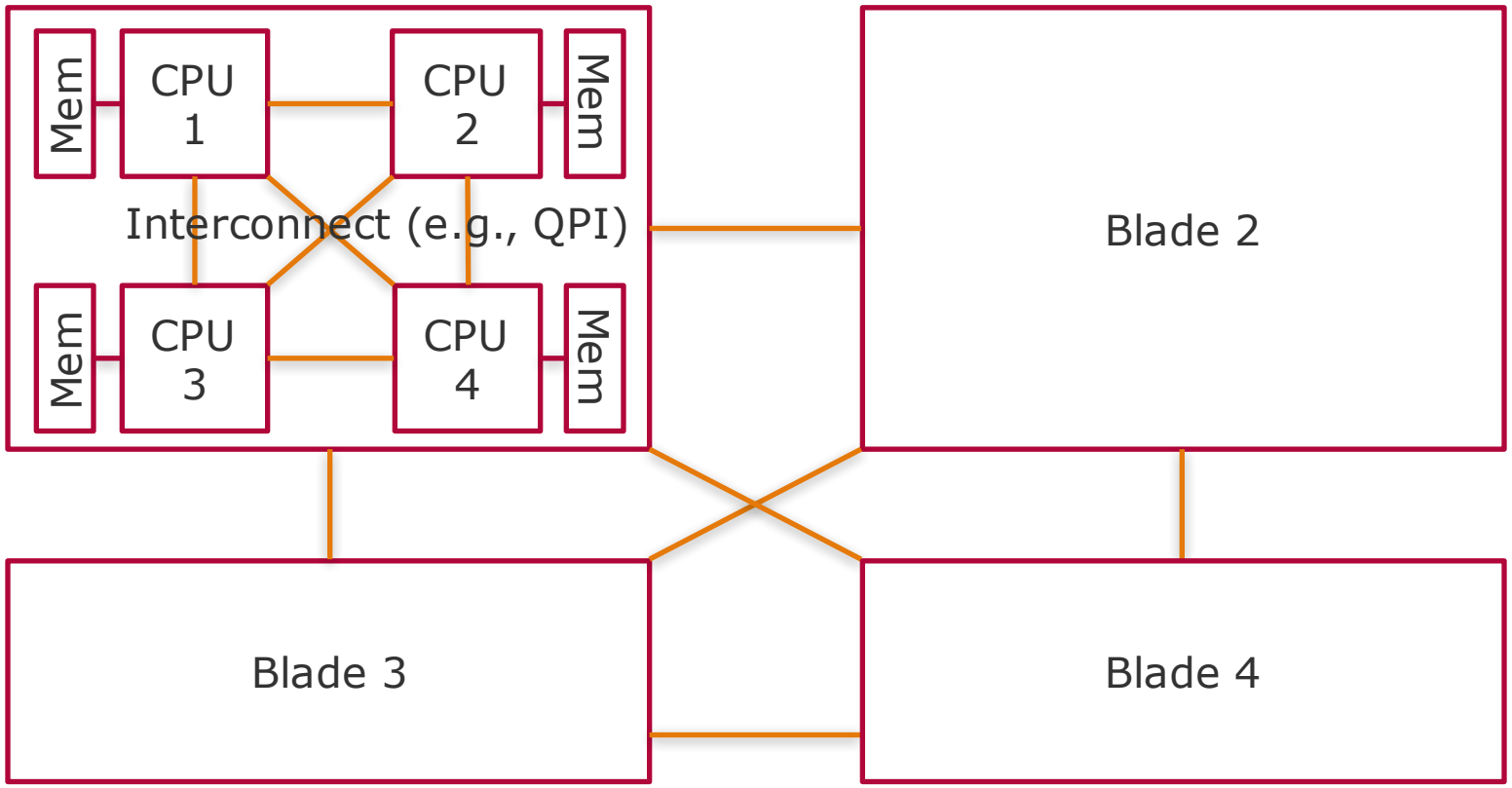
Benefits

- Chunks are stable once they are compressed
- Simplified data placement in general and especially on NUMA systems
- Enhanced query execution

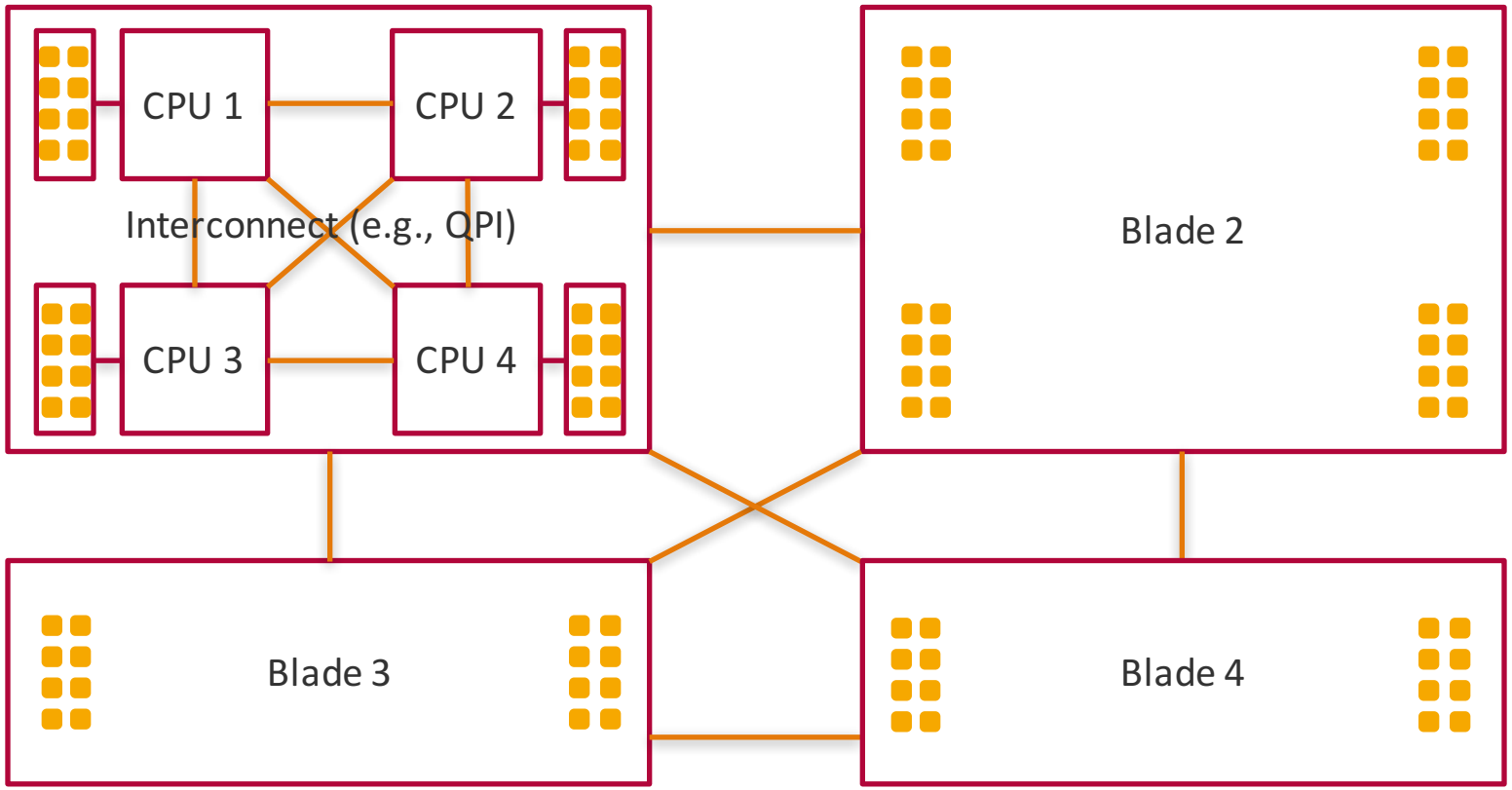
Drawbacks

- Potentially increased memory consumption by duplicated meta data structures

Chunks – Non-Uniform Memory Access



Chunks – Non-Uniform Memory Access



Experiments with Chunks

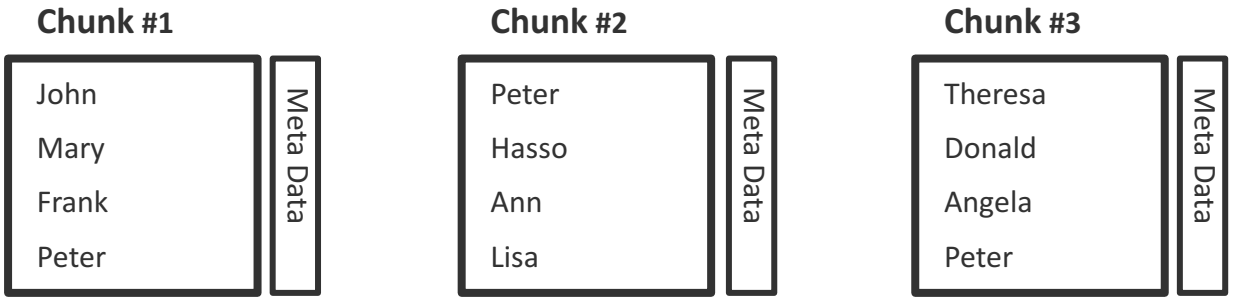
Benefits

- Chunks are stable once they are compressed
- Simplified data placement in general and especially on NUMA systems
- Enhanced query execution

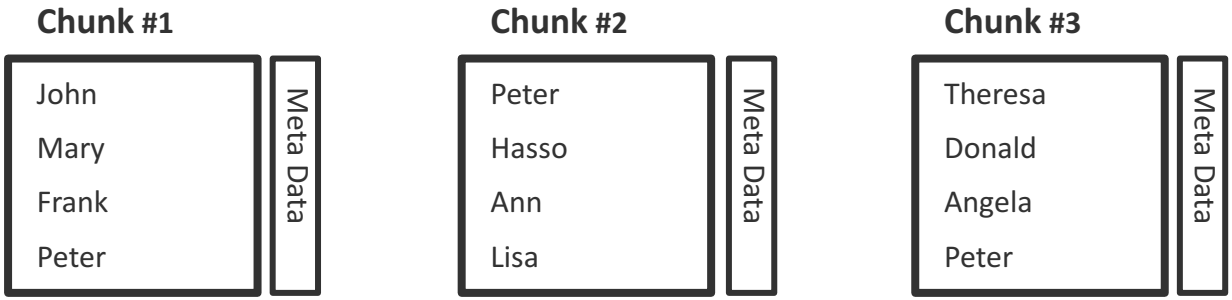
Drawbacks

- Potentially increased memory consumption by duplicated meta data structures

Chunks – Enhanced Query Execution

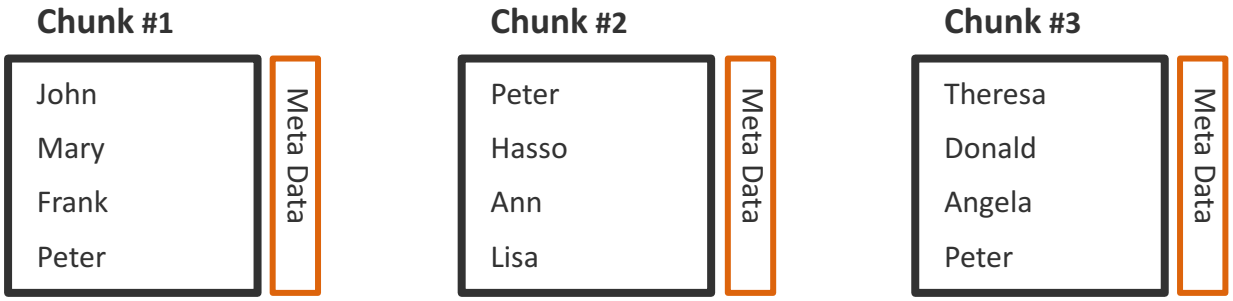


Chunks – Enhanced Query Execution



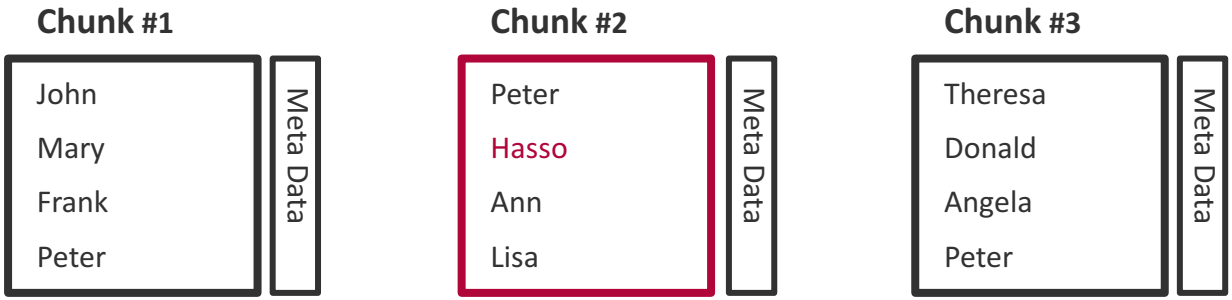
SELECT * FROM customers WHERE firstname = 'Hasso'

Chunks – Enhanced Query Execution



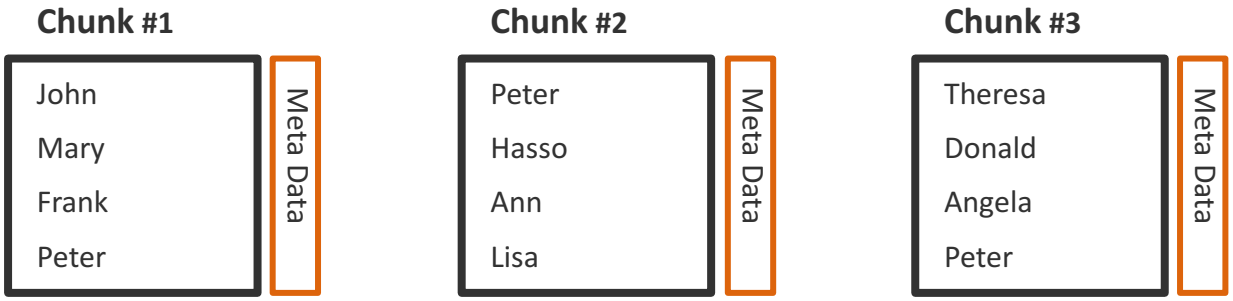
SELECT * FROM customers WHERE firstname = 'Hasso'

Chunks – Enhanced Query Execution



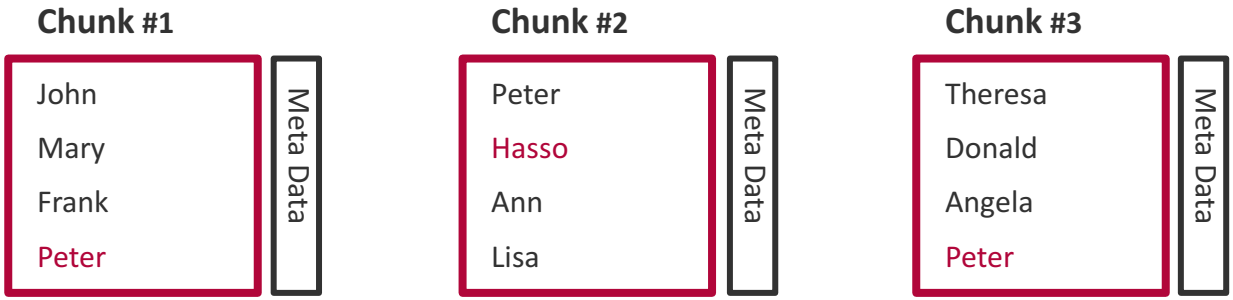
SELECT * FROM customers WHERE firstname = 'Hasso'

Chunks – Enhanced Query Execution



SELECT * FROM customers WHERE firstname = 'Peter'

Chunks – Enhanced Query Execution



SELECT * FROM customers WHERE firstname = 'Peter'

Experiments with Chunks

Benefits

- Chunks are stable once they are compressed
- Simplified data placement in general and especially on NUMA systems
- Enhanced query execution

Drawbacks

- Potentially increased memory consumption by duplicated meta data structures

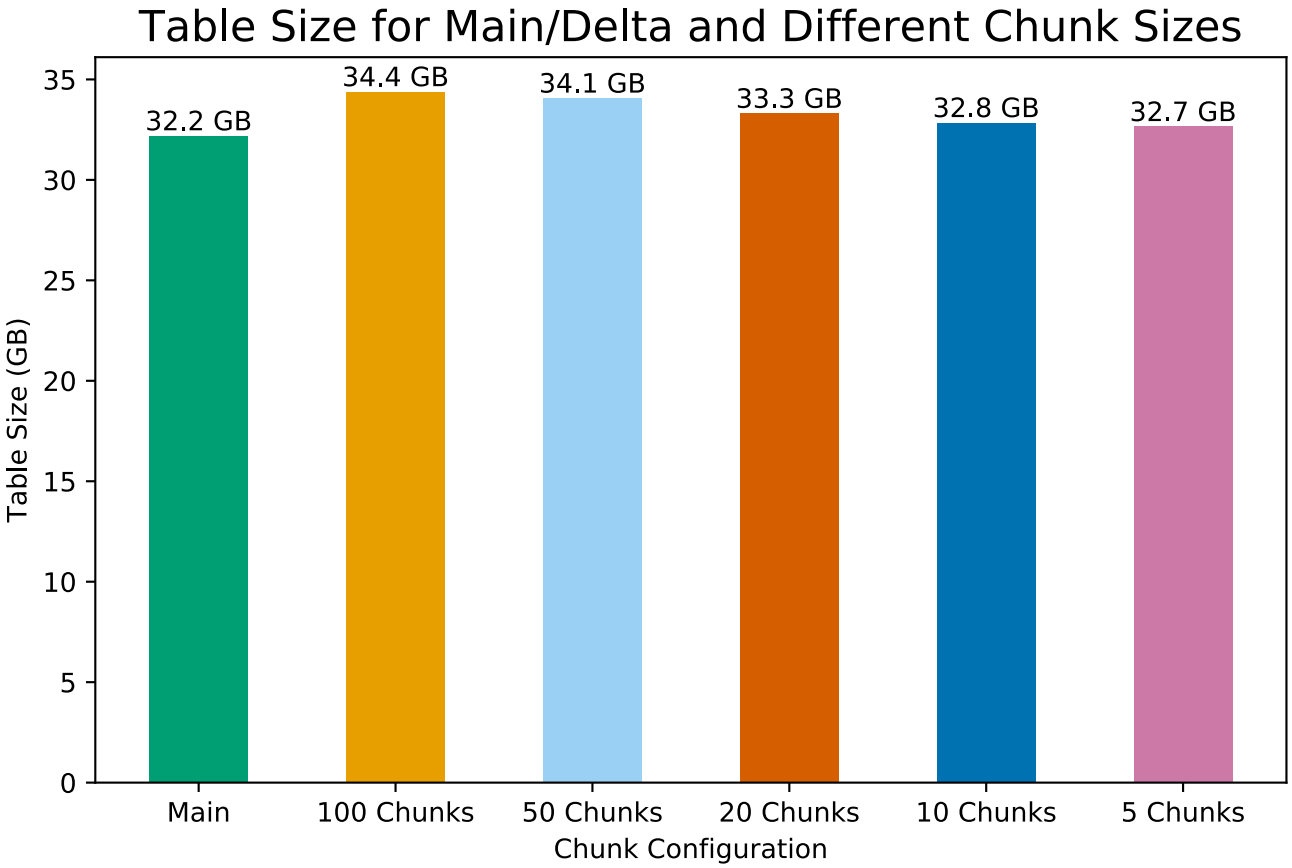
Experiments with Chunks

1. How does chunking affect the memory footprint?
2. If the last chunk is uncompressed, what is its effect on the total memory consumption?
3. What are performance implications of chunks?
    1. Considering single-threaded and multi-threaded (NUMA) execution

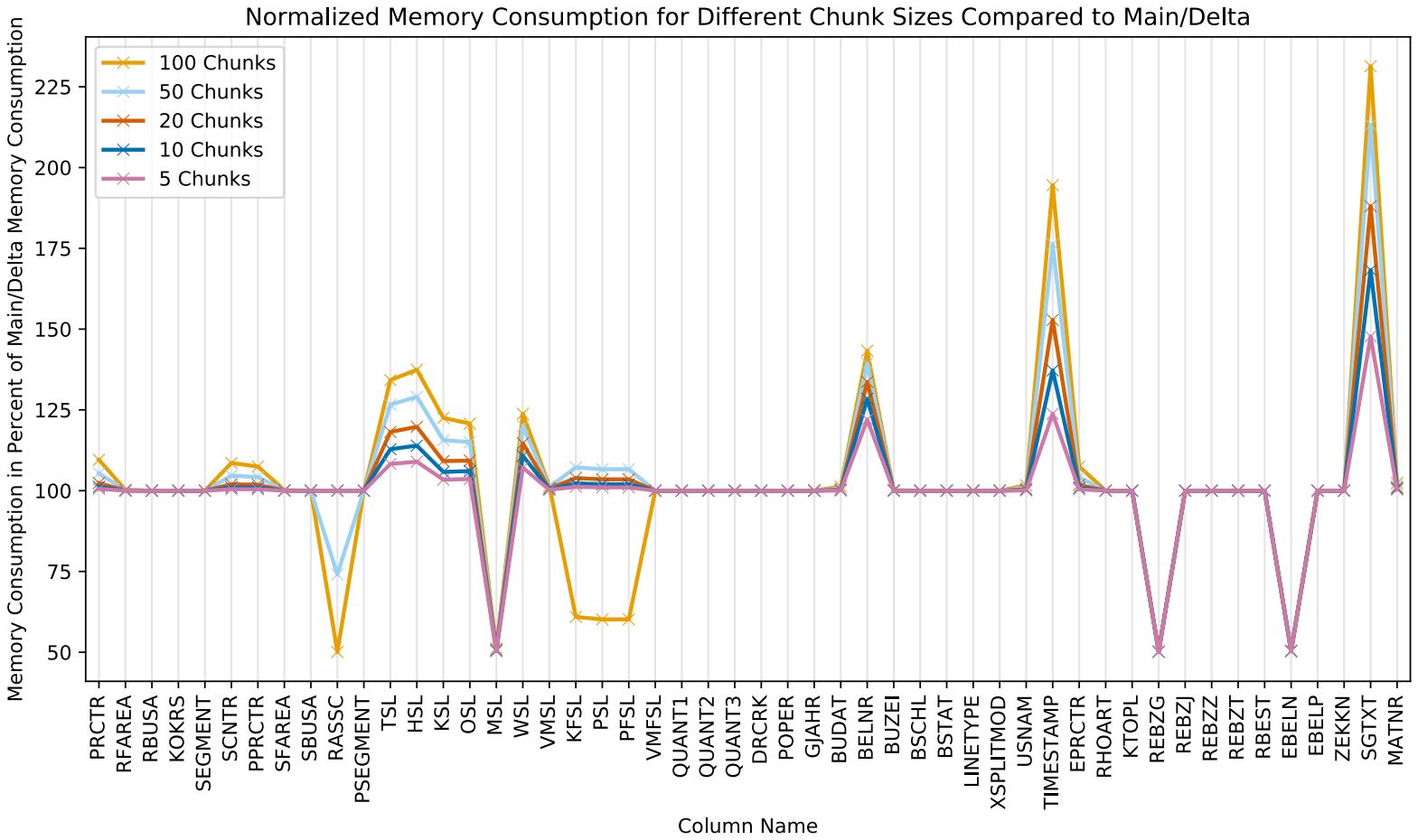
Experiments with Chunks

- Methodology:
  - Utilize real-world data from a productive SAP system
  - Extract actual queries from system‘s plan cache
  - Load 100M rows of data into Opossum/Hyrise
  - Measure memory footprint/query performance
  - Repeat experiments for different chunk sizes

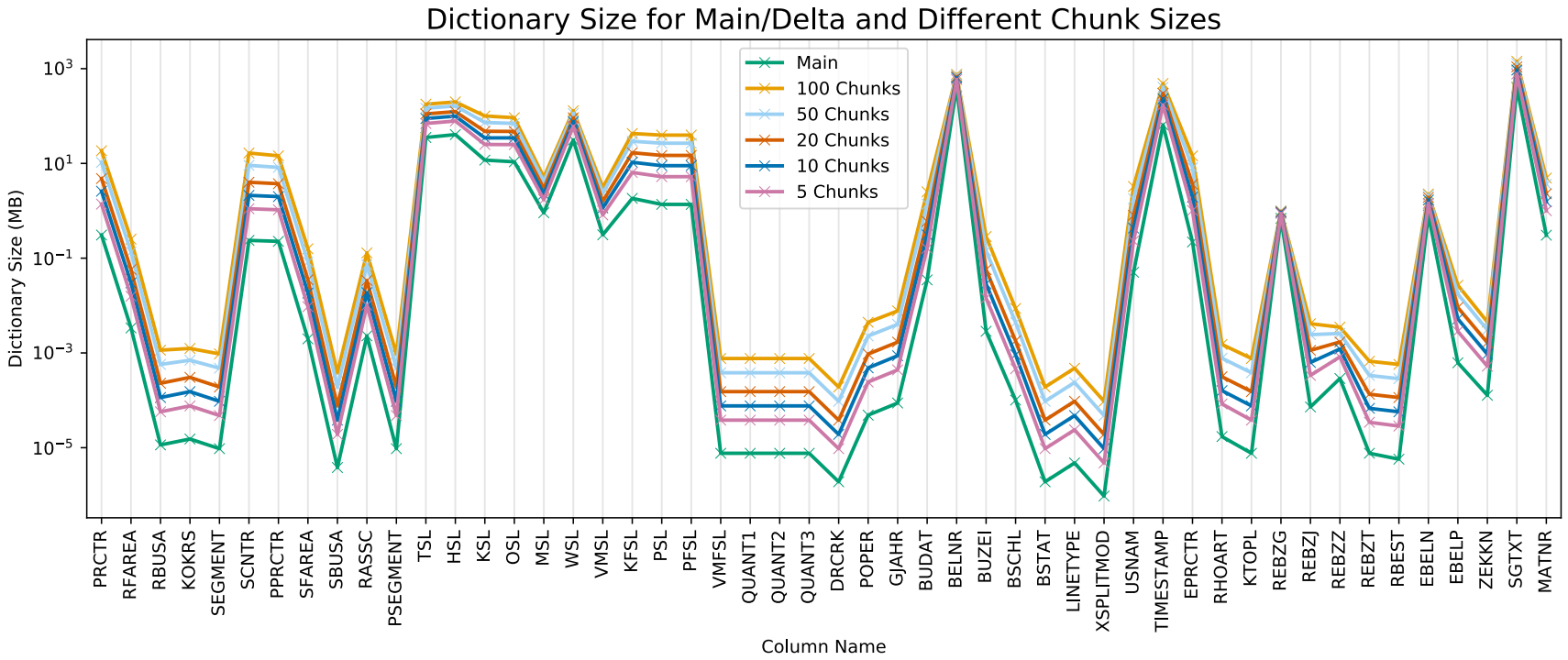
Experiments with Chunks: Memory Footprint



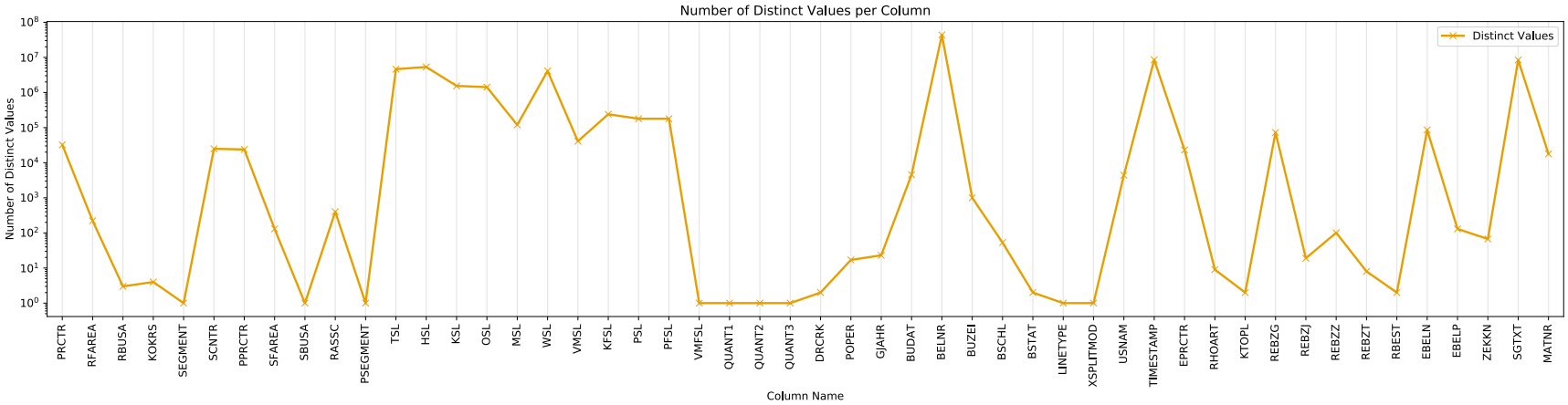
Experiments with Chunks: Memory Footprint



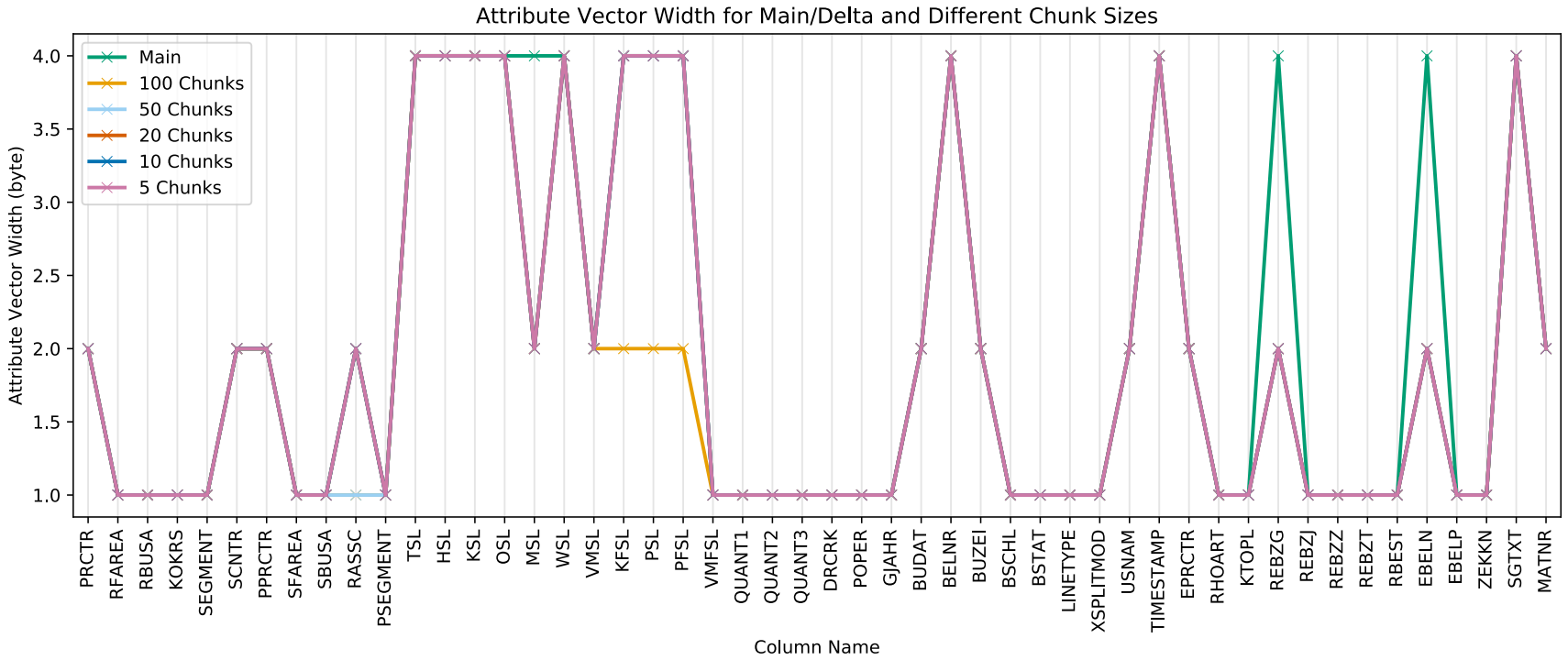
Experiments with Chunks: Memory Footprint



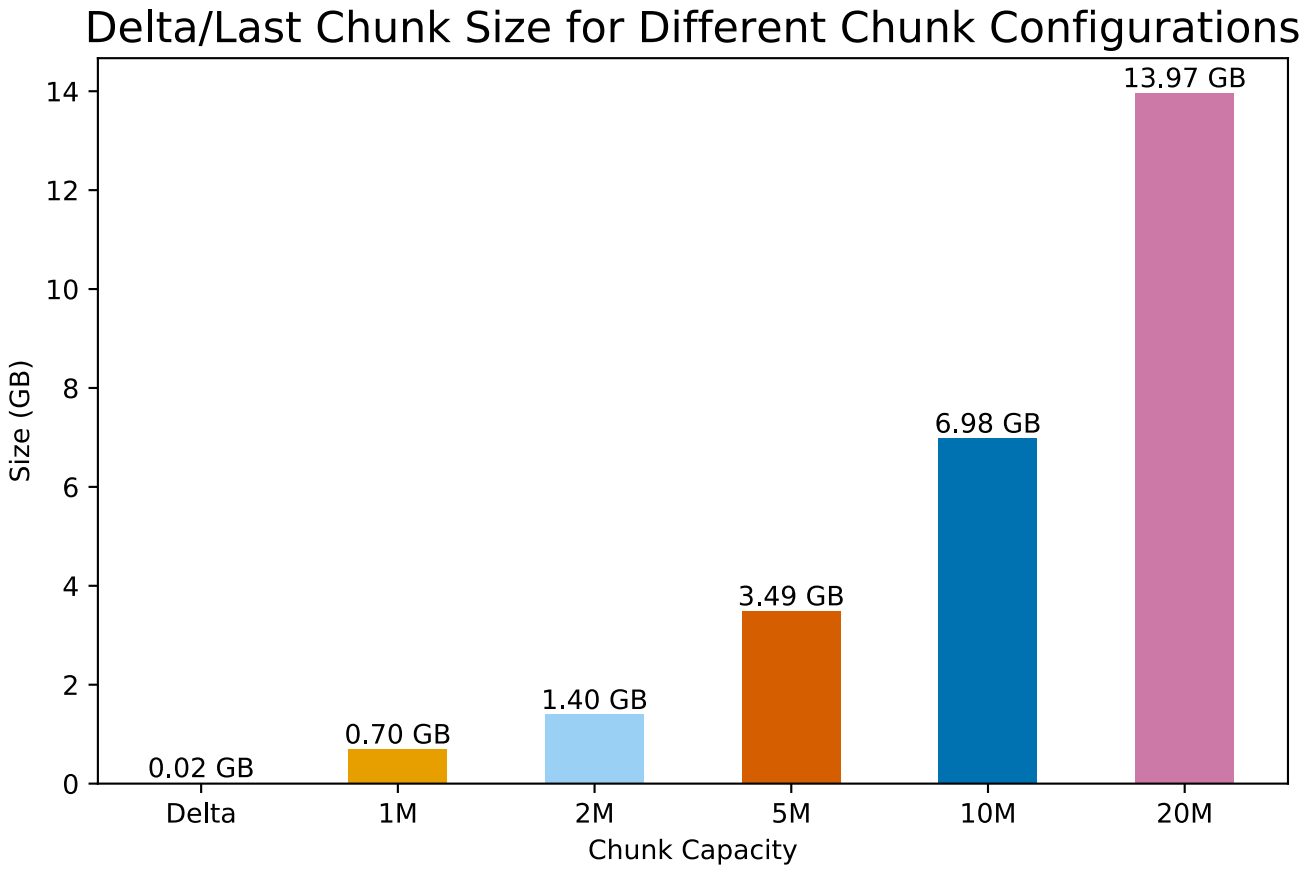
Experiments with Chunks: Memory Footprint



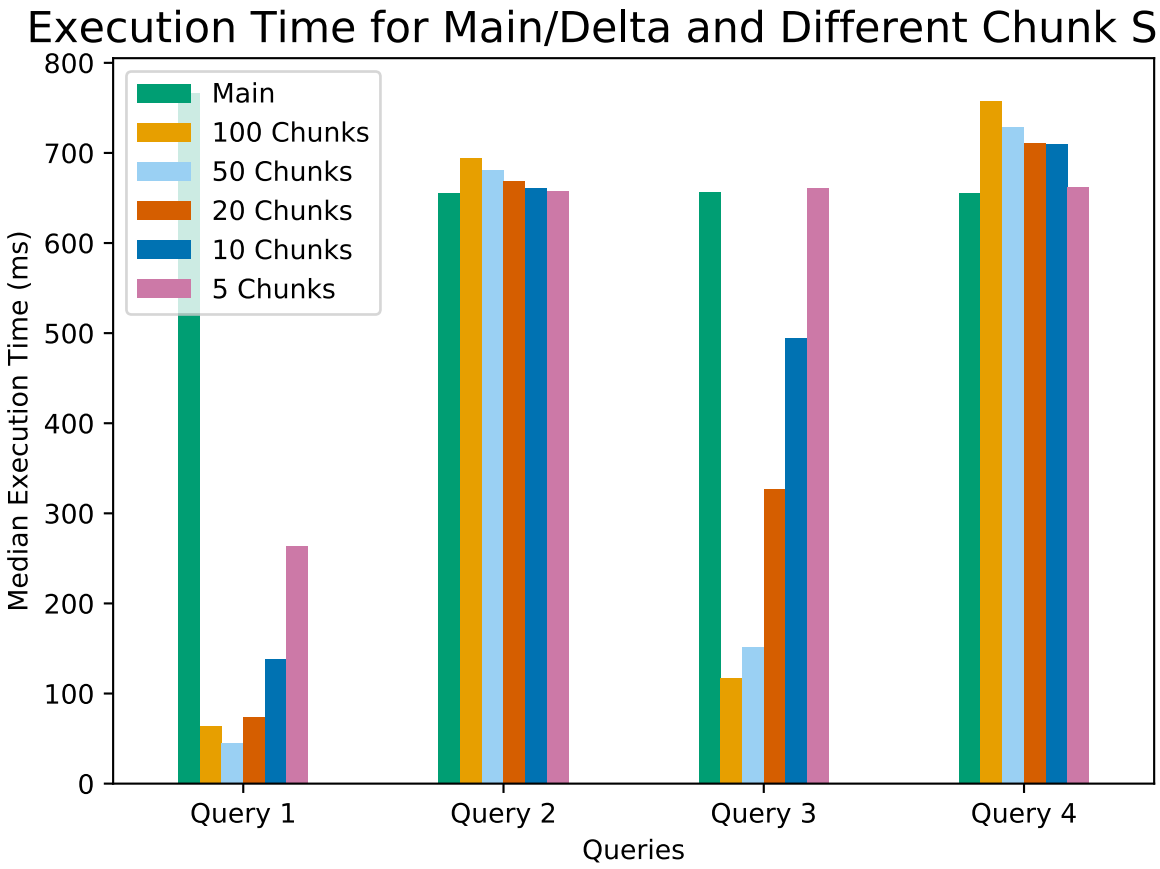
Experiments with Chunks: Memory Footprint



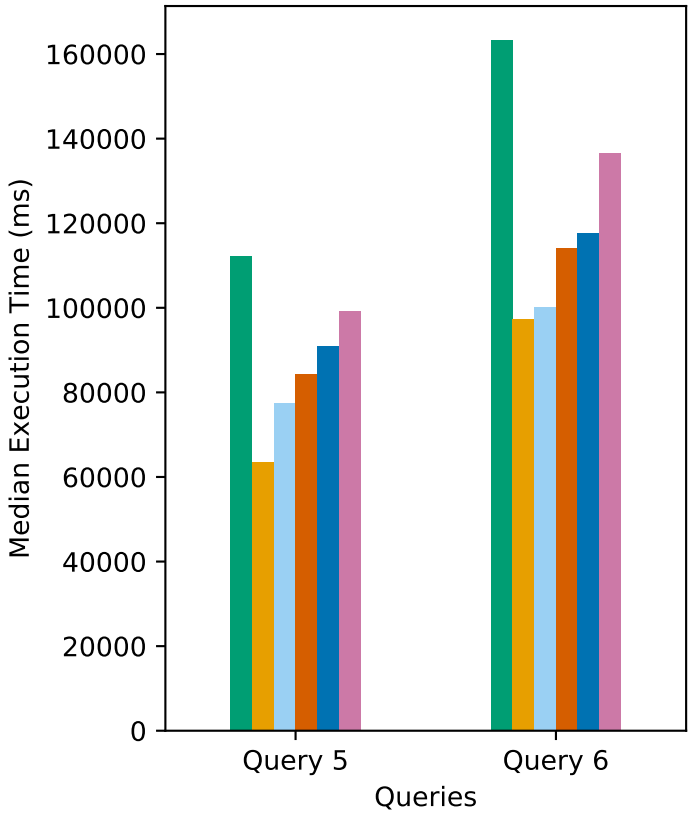
Experiments with Chunks: Memory Footprint



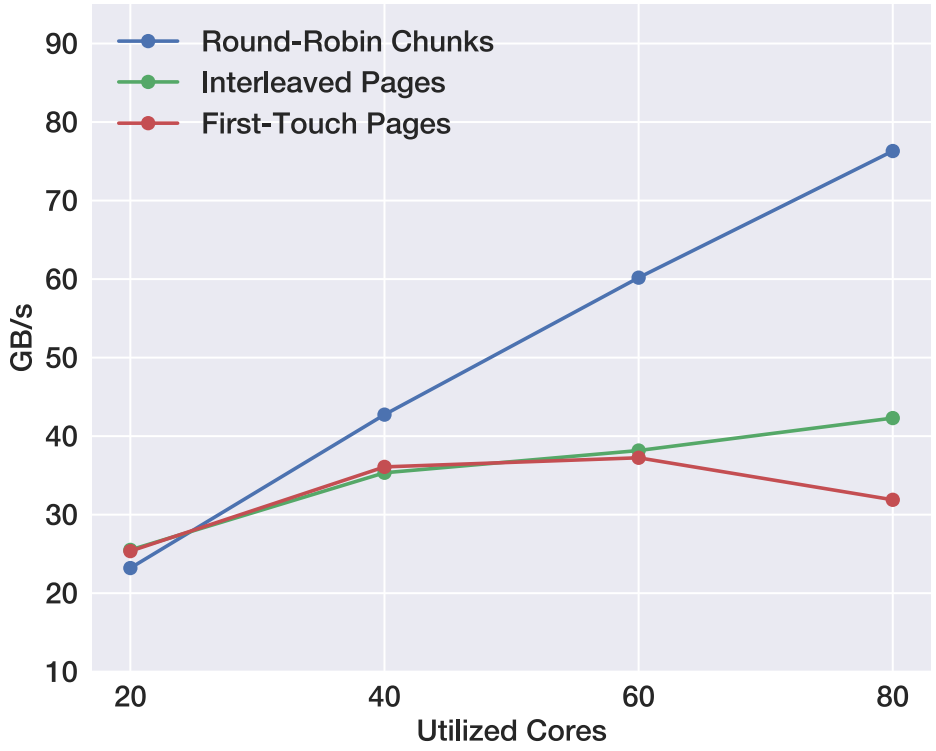
Experiments with Chunks: Performance (single-threaded)



Experiments with Chunks: Performance (single-threaded)



Experiments with Chunks: Performance (NUMA)



Optionals

- *„Manages an optional contained value, i.e. a value that may or may not be present.“*
- Example use case: A table scan that supports between and, therefore, needs two search value parameters
- Syntax:

```
#include <optional>
// Templated object of type std::optional<T>
std::optional<AllTypeVariant> opt;
std::optional<AllTypeVariant> opt2 = std::nullopt;
std::optional<AllTypeVariant> opt3 = 17;
if (opt) {
    do_something(*opt);
}
```

Optionals

Any ideas how to implement that?

~~**std**::pair<T,bool>~~

```
template <typename T>
class optional {
  bool _initialized;
  T _storage;
};
```

What is the result of `sizeof(std::optional<uint32_t>)`?

Lambda Expressions

A simplified table scan…

```
for (auto i = 0; i < value_column.size(); ++i) {
  switch (_scan_type) {
    case ScanType::OpEquals: {
      return value_column.get(i) == search_value;
      break;
    }
    case ScanType::OpNotEquals: {
      return value_column.get(i) != search_value;
      break;
    }
    case ScanType::OpLessThan: {
      return value_column.get(i) < search_value;
      break;
// [...]
```

Lambda Expressions

With lambda expressions

```
auto comparator = get_comparator(_scan_type);
for (auto i = 0; i < value_column.size(); ++i) {
  return comparator(value_column.get(i), search_value);
}
```

```
auto get_comparator(ScanType type) {
  switch (type) {
    case ScanType::OpEquals: {
      _return = [](auto left, auto right) { return left == right; };
      break;
    }
    case ScanType::OpNotEquals: {
      _return = [](auto left, auto right) { return left != right; };
      break;
    }
    // [...]
  }
}
```

+ separation of concerns
+ checks only once
+ reuse

Lambda Expressions

Syntax:

```
auto f = [ captures ] ( params ) -> ret { body };
```

Code goes here

Variables that you take from the current scope

Return value of the lambda (if you leave it out, the compiler does it for you)

Can store lambdas in variables (and even members)

Parameters that are passed when the lambda is called

You must use auto here

Lambda Expressions

```
auto f = [ captures ] ( params ) -> ret { body };
```

```
int main() {
  auto f = []() {
    std::cout << "Hallo Welt" << std::endl;
  };

  f();
}
```

Lambda Expressions

```
auto f = [ captures ] ( params ) -> ret { body };
```

```
int main() {
  auto f = [](const std::string& name) {
    std::cout << "Hallo " << name << std::endl;
  };

  f("Alexander");
}
```

Lambda Expressions

```
auto f = [ captures ] ( params ) -> ret { body };
```

```
int main() {
  std::string my_name{"Larry"};

  auto f = [my_name](const std::string& name) {
    std::cout << "Hallo " << name << ", ich bin "
              << my_name << std::endl;
  };

  f("Alexander");
}
```

Lambda Expressions

```
auto f = [ captures ] ( params ) -> ret { body };
```

```
int main() {
  std::string my_name{"Larry"};

  auto f = [&my_name](const std::string& name) {
    std::cout << "Hallo " << name << ", ich bin "
              << my_name << std::endl;
  };

  f("Alexander");
}
```

Lambda Expressions

```
auto get_lambda() {
  std::string my_name{"Larry"};
  return [my_name]() {
    std::cout << "Ich bin " << my_name << std::endl;
  };
}

int main() {
  f = get_lambda();

  // my_name is undefined here

  f();
}
```

Lambda Expressions

```
auto get_lambda() {
  std::string my_name{"Larry"};

  return [&my_name]() {
    std::cout << "Ich bin " << my_name << std::endl;
  };
}

int main() {
  f = get_lambda();

  // my_name is undefined here

  f();
}
```

Lambda Expressions

User code

```
int main()
{
  vector<X> v;

  // Add elements to the vector…

  int total = 0;
  int offset = 1;

  for_each(v.begin(), v.end(),
    [&total, offset](X& elem) { total += elem.getVal() + offset; });

  cout << total << endl;
}
```

From https://blog.feabhas.com/2014/03/demystifying-c-lambdas/
A great resource if you want to learn more about lambdas

Lambda Expressions

```
for_each(v.begin(), v.end(),
  [&total, offset](X& elem) { total += elem.getVal() + offset; });
```

```
for_each(v.begin(), v.end(), _SomeCompilerGeneratedName_{total, offset});
```

Compiler generated (conceptual)

```
class _SomeCompilerGeneratedName_
{
public:
  _SomeCompilerGeneratedName_(int& t, int o) : total_{t}, offset_{o} {}
  void operator() (X& elem) const {total_ += elem.getVal() + offset_;}

private:
  int& total_;      // Context captured by reference
  int  offset_;     // Context captured by value
};
```

Next Week

- Relational Algebra
- Operators
- Presentation of Sprint 3