WEEK 3

BYOD
AGENDA

▸ Dictionary Encoding
▸ Organization
▸ Sprint 3
Memory access is the new bottleneck
Decrease number of bits used for data representation
Dictionary encoding is an “easy-to-implement” fixed-width compression and basis for several other compression techniques.

Idea: encode every distinct value of a vector (large) with a distinct fixed-length integer value (small).
### Dictionary Encoding - Example: Sample Data

- World population: 8 billion records

<table>
<thead>
<tr>
<th>recID</th>
<th>fname</th>
<th>lname</th>
<th>gender</th>
<th>city</th>
<th>country</th>
<th>birthday</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>39</td>
<td>John</td>
<td>Smith</td>
<td>m</td>
<td>Chicago</td>
<td>USA</td>
<td>12.03.1964</td>
</tr>
<tr>
<td>40</td>
<td>Mary</td>
<td>Brown</td>
<td>f</td>
<td>London</td>
<td>UK</td>
<td>12.05.1964</td>
</tr>
<tr>
<td>41</td>
<td>Jane</td>
<td>Doe</td>
<td>f</td>
<td>Palo Alto</td>
<td>USA</td>
<td>23.04.1976</td>
</tr>
<tr>
<td>42</td>
<td>John</td>
<td>Doe</td>
<td>m</td>
<td>Palo Alto</td>
<td>USA</td>
<td>17.06.1952</td>
</tr>
<tr>
<td>43</td>
<td>Peter</td>
<td>Schmidt</td>
<td>m</td>
<td>Potsdam</td>
<td>GER</td>
<td>11.11.1975</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table: `world_population`
Dictionary stores all distinct values with an implicit valueID

Attribute vector stores valueIDs for all entries in the column (positions are stored implicitly)
DICTIONARY ENCODING - EXAMPLE: COMPRESSION RATE

- 5 million distinct values, all have a size of 50 B
  - Bits required per valueID: \( \text{ceil}(\log_2(5,000,000)) \) b = 23
  - Dictionary size: \( 5 \times 10^6 \times 50 \text{ B} = 250 \times 10^6 \text{ B} = 0.250 \text{ GB} \)
  - Attribute vector size: \( 8 \times 10^9 \times 23\text{b} = 23 \times 10^9 \text{ B} = 23 \text{ GB} \)
  - Uncompressed: \( 8 \times 10^9 \times 50 \text{ B} = 400 \times 10^9 \text{ B} = 400 \text{ GB} \)

- Compression rate = uncompressed size / compressed size
  = \( 400\text{GB} / (23 \text{ GB} + 0.250 \text{ GB}) \approx 17 \)
**Retrieve all persons (recIDs) with name “Mary”**

1. **Search valueID for “Mary” (requested value)**

2. **Scan Attribute vector for “24” (found valueID)**
DICTIONARY ENCODING – SORTED DICTIONARY: ADVANTAGES

- Dictionary entries are sorted by their value
  - Dictionary search complexity: $O(\log(n))$ instead $O(n)$
  - Speed up range queries
- Dictionary entries can be further compressed
Dictionary entries are sorted by their value

- Resorting for every new value that does not belong to the end of the sorted sequence (relatively cheap)
- Updating the attribute vector (costly)

Dictionary adds additional indirection for materialization

Overhead for large number of data modifying operations
Dictionary encoding is applied to full chunk

Sorted dictionaries are used

valueIDs are the C99 fixed-width unsigned integer types (http://en.cppreference.com/w/c/types/integer)
QUESTIONS
First sprint was due yesterday

Document and interfaces for 2nd sprint will be released today

Use piazza for questions and discussions

Have fun compressing the opossum
Build your own Database

Week 3
Outlook

1. Review Sprint 1
2. Move Constructors
3. `std::move`
4. `(?:g?l|p?r|x)values`
All groups submitted a (mostly) working implementation on time 😊
Error Handling

- Some groups check for most edge cases, others do not
- We have no standard rules for error handling so far

```cpp
void StorageManager::add_table(const std::string &name,
    std::shared_ptr<Table> table) { _tables[name] = table; }
```

- New policy: Always do checks when they are (almost) free, especially when they are in the control path (not the per-row data path)
- Use IS_DEBUG for expensive checks
Error Handling

- Most STL-Containers can help us a lot at almost zero cost

```cpp
std::map<std::string, Table> _tables;

A _tables[name] = table;

B if(_tables.find(name) != _tables.end()) {
    _tables[name] = table;
}

C _tables.insert({name, table});

D auto r = _tables.insert({name, table});
if(!r.second) throw std::runtime_error("...");
```
Error Handling

```cpp
std::vector<Chunk> _chunks;

A
Chunk &Table::get_chunk(ChunkID chunk_id) {
    return _chunks.at(chunk_id);
}

B
Chunk &Table::get_chunk(ChunkID chunk_id) {
    return _chunks[chunk_id];
}

C
Chunk &Table::get_chunk(ChunkID chunk_id) {
    if(chunk_id >= _chunks.size())
        throw std::runtime_error(...)
    return _chunks.at(chunk_id);
}
```
Error Handling

• What can we improve about this code?

```cpp
std::map<std::string, Table> _tables;
if (_tables.find(name) != _tables.end()) {
    _tables.erase(name);
}
```

3) Removes the element (if one exists) with the key equivalent to key.

```cpp
size_type erase( const key_type& key );
```

```cpp
std::map<std::string, Table> _tables;
_tables.erase(name);
```
• How can we further improve this?

```cpp
std::map<std::string, Table> _tables;

_tables.erase(name);
```

**Return value**

3) Number of elements removed.

```cpp
std::map<std::string, Table> _tables;

if(!_tables.erase(name)) {
    throw std::runtime_error(...)
}
```
Initializer Lists

• What is the problem with this code?

```cpp
class Table {
    Table(const size_t chunk_size) {
        _chunk_size = chunk_size;
        _chunks.push_back(std::make_shared<Chunk>());
    }

protected:
    size_t _chunk_size;
    [...]}
};
```
Initializer Lists

class Table {
    Table(const size_t chunk_size) {
        _chunk_size = chunk_size;
        _chunks.push_back(std::make_shared<Chunk>());
    }

protected:
    const size_t _chunk_size;
 [...]
Initializer Lists

- It is better to initialize members in the constructor’s initialization list

```cpp
class Table {
    Table(const size_t chunk_size) : _chunk_size(chunk_size) {
        _chunks.push_back(std::make_shared<Chunk>());
    }

protected:
    const size_t _chunk_size;
    [...]}
```
Initializer Lists

```cpp
struct A {
    A() { std::cout << "const A" << std::endl; }
    A(int x) { std::cout << "const A: " << x << std::endl; }
    ~A() { std::cout << "dest A" << std::endl; }
};

struct B {
    A a;
    B(int x) { a = A(x); }
};

struct C {
    A a;
    C(int x) : a(x) {}
};

int main() {
    B(1);
    C(2);
}
```

![Output of the program showing the expected output: const A, const A: 1, dest A, const A: 2, dest A.](image)
“std::vector<Chunk> might re-locate the memory where a Chunk lives on, e.g. push_back(). So the memory location's persistance is NOT guaranteed and using the reference might be unsafe if further operations are performed on the Table, [...]”
Use reference wherever you can, pointers wherever you must.

Avoid pointers until you can't.

The reason is that pointers make things harder to follow/read, less safe and far more dangerous manipulations than any other constructs.

So the rule of thumb is to use pointers only if there is no other choice.

A smart pointer is a class that wraps a 'raw' (or 'bare') C++ pointer, to manage the lifetime of the object being pointed to. There is no single smart pointer type, but all of them try to abstract a raw pointer in a practical way.

Smart pointers should be preferred over raw pointers. If you feel you need to use pointers (first consider if you really do), you would normally want to use a smart pointer as this can alleviate many of the problems with raw pointers, mainly forgetting to delete the object and leaking memory.
**Chunk: Pointer vs Reference**

**Iterator validity**

If a reallocation happens, all iterators, pointers and references related to the container are invalidated. Otherwise, only the end iterator is invalidated, and all iterators, pointers and references to elements are guaranteed to keep referring to the same elements they were referring to before the call.

- In this case:
  - Modifying a table while processing it as part of a query does not happen
  - In multi-threaded environments, modifications to `std::vectors` cause more problems anyway
Chunk: Pointer vs Reference
```cpp
class Chunk {
    public:
        Chunk() {}
};

class Table {
    private:
        void addNewChunk();
}

void Table::append(std::initializer_list<AllTypeVariant> values) {
    if (_chunks.back()->size() == _chunk_size) append_new_chunk();
    _chunks.back()->append(values);
}
```
• Not a problem now, but might become in the future...

```cpp
void Table::append(std::initializer_list<AllTypeVariant> values) {
    if (_chunk_size != 0 && _chunks.back().size() >= _chunk_size) {
        _chunks.push_back(Chunk());
        for (auto type_it = _column_types.begin(); type_it != _column_types.end(); type_it++) {
            auto column = make_shared_by_column_type<BaseColumn, ValueColumn>(*type_it);
            _chunks.back().add_column(column);
        }
    }
    _chunks.back().append(values);
}
```
Miscellaneous

• Not a problem now, but might become in the future...

```cpp
void Table::append(std::initializer_list<AllTypeVariant> values) {
    if (_chunk_size != 0 && _chunks.back().size() >= _chunk_size) {
        Chunk new_chunk;
        for (auto type_it = _column_types.begin(); type_it != _column_types.end(); type_it++) {
            auto column = make_shared_by_column_type<BaseColumn, ValueColumn>(*type_it);
            new_chunk.add_column(column);
        }
        _chunks.emplace_back(std::move(new_chunk));
    }
    _chunks.back().append(values);
}
```
• Make the code shorter

```cpp
void Table::append(std::initializer_list<AllTypeVariant> values) {
    if (_chunk_size != 0 && _chunks.back().size() >= _chunk_size) {
        Chunk new_chunk;
        for (auto&& type : _column_types) {
            auto column = make_shared_by_column_type<BaseColumn, ValueColumn>(type);
            new_chunk.add_column(column);
        }
        _chunks.emplace_back(std::move(new_chunk));
    }
    _chunks.back().append(values);
}
```
Deleting Copy Constructors
Deleting Copy Constructors

• Big classes in our database should not be copyable
• Deleted copy constructors should be default

• In this sprint: base_column
Avoiding Copies

• We want to avoid unnecessary copies as much as possible

• (Some copies make sense – most times, there is no point passing an integer by reference)

• How does the compiler know when to avoid copies and how can we help?
Avoiding Copies for a String

class string {
    char *buf;

public:
    string(const char *str) {
        size_t size = strlen(str) + 1;
        buf = (char*)malloc(size);
        memcpy(buf, str, size);
    }
    void print() { std::cout <<
    };

Rule of Three
Destructor
Copy Const
Copy Assign
Avoiding Copies for a String

class string {
    char *buf;
public:
    string(const char *str) {
        size_t size = strlen(str) + 1;
        buf = (char*)malloc(size);
        memcpy(buf, str, size);
    }
    ~string() { free(buf); }
    string(const string& that) {
        size_t size = strlen(that.buf) + 1;
        buf = (char*)malloc(size);
        memcpy(buf, that.buf, size);
    }
    string& operator=(const string &that) { [...]
    }
    void print() { std::cout << buf << std::endl; }
};
Avoiding Copies for a String

class string {
    char *buf;
public:
    string(const char *str) {
        size_t size = strlen(str) + 1;
        buf = (char*)malloc(size);
        memcpy(buf, str, size);
        std::cout << "allocated " << size << " bytes" << std::endl;
    }
    ~string() { free(buf); }
    string(const string& that) {
        size_t size = strlen(that.buf) + 1;
        buf = (char*)malloc(size);
        memcpy(buf, that.buf, size);
        std::cout << "allocated " << size << " bytes" << std::endl;
    }
    string& operator=(const string &that) {[[...]]}
    void print() { std::cout << buf << std::endl; }
};
Avoiding Copies for a String

```cpp
int main() {
    string a("test");
    a.print();

    string b(a);
    b.print();

    string c = a;
    c.print();
}
```

```
[∶~/Desktop/tmp] $ g++-6 test.cpp -std=c++03 -Wall -Wextra && .a.out
allocated 5 bytes
allocated 5 bytes
test
allocated 5 bytes
test
allocated 5 bytes
test
```
Avoiding Copies for a String

```cpp
// I also modified the print statements to print the string

int main() {
    std::vector<std::string> v;
    v.push_back("test");
}
```

```
$ g++-6 test.cpp -std=c++1z -Wall -Wextra && ./a.out
allocated 5 bytes for "test"
allocated 5 bytes for "test"
```
Avoiding Copies for a String

// I also modified the print statements to print the string

int main() {
    std::vector<string> v;
    v.push_back("foo");
    v.push_back("bar");
}

allocated 4 bytes for "foo" (constructor)
allocated 4 bytes for "foo" (copy constructor)
allocated 4 bytes for "bar" (constructor)
allocated 4 bytes for "bar" (copy constructor)
allocated 4 bytes for "foo" (copy constructor)
Avoiding Copies for a String

„The purpose of a move constructor is to steal as many resources as it can from the original object, as fast as possible, because the original does not need to have a meaningful value any more, because it is going to be destroyed (or sometimes assigned to) in a moment anyway.“

Avoiding Copies for a String

```cpp
class string {
    [...]  
    string(string&& that) : buf(that.buf) {
        that.buf = nullptr;
        std::cout << "moved " << buf << std::endl;
    }
};
```

Rule of Five
Avoiding Copies for a String

class string {
    [...]  
    string(string&& that) : buf(that.buf) {
        that.buf = nullptr;
        std::cout << "moved " << buf << std::endl;
    }  
    string& operator=(string&& that) {
        free(buf);
        buf = that.buf;
        that.buf = nullptr;
        std::cout << "moved " << buf << std::endl;
        return *this;
    }
};
Avoiding Copies for a String

```cpp
string get_test() {
    return string("test");
}

int main() {
    std::vector<string> v;
    v.push_back("foo");
    v.push_back(get_test());
}
```

```
[:~/Desktop/tmp] $ g++-6 test.cpp -std=c++1z -Wall -Wextra && ./a.out
allocated 4 bytes for "foo" (constructor)
moved foo
allocated 5 bytes for "test" (constructor)
moved test
allocated 4 bytes for "foo" (copy constructor)
```
Avoiding Copies for a String

class string {
    [...] 
    string(string&& that) noexcept : buf(that.buf) {
        that.buf = nullptr;
        std::cout << "moved " << buf << std::endl;
    }
    string& operator=(string&& that) noexcept {
        free(buf);
        buf = that.buf;
        that.buf = nullptr;
        std::cout << "moved " << buf << std::endl;
        return *this;
    }
};

If a search for a matching exception handler leaves a function marked `noexcept` or `noexcept(true)`, `std::terminate` is called immediately.
Avoiding Copies for a String

```cpp
int main() {
    string a("baz");
    std::vector<string> v;
    v.push_back(a);

    // we'll never use a again...
}
```

```
[~:/Desktop/tmp] $ g++-6 test.cpp -std=c++1z -Wall -Wextra && ./a.out
allocated 4 bytes for "baz" (constructor)
allocated 4 bytes for "baz" (copy constructor)
```
Avoiding Copies for a String

```cpp
int main() {
    string a("baz");
    std::vector<string> v;

    v.push_back(std::move(a));

    // we'll never use a again...
}
```

```bash
[:~/Desktop/tmp] $ g++-6 test.cpp -std=c++1z -Wall -Wextra && ./a.out
allocated 4 bytes for "baz" (constructor)
moved baz
```
Avoiding Copies for a String

```cpp
int main() {
    string a("baz");
    std::vector<string> v;

    v.push_back(std::move(a));

    // we'll never use a again...

    string b(a);
    // but you promised :(
}
```

```
[~/Desktop/tmp] $ g++-6 test.cpp -std=c++1z -Wall -Wextra -O3 && ./a.out
allocated 4 bytes for "baz" (constructor)
moved baz
Segmentation fault: 11
```
What is std::move?

- What does std::move do?
- From an instruction POV: Nothing
- „std::move is used to *indicate* that an object t may be "moved from", i.e. allowing the efficient transfer of resources from t to another object.
- „In particular, std::move produces an *xvalue expression* that identifies its argument t. It is exactly equivalent to a static_cast to an rvalue reference type."

```cpp
template <typename T>
typename remove_reference<T>::type&& move(T&& arg) {
    return static_cast<typename remove_reference<T>::type&&>(arg);
}
```
Deep Dive: l,r,gl,pr,x,wtfvalues

```
[::~/Desktop/tmp] 1 $ g++-6 test.cpp -std=c++1z -Wall -Wextra -O3 & & ./a.out

 test.cpp: In function 'int main()':
 test.cpp:61:11: error: lvalue required as left operand of assignment
   zwei() = 3;
   ^

[::~/Desktop/tmp] 1 $ g++-6 test.cpp -std=c++1z -Wall -Wextra -O3 & & ./a.out

 test.cpp: In function 'int& zwei()':
 test.cpp:58:22: error: invalid initialization of non-const reference of type 'int&' from an rvalue of type 'int'
   int &zwei() { return 2;}
   ^
```
Deep Dive: l,r,gl,pr,x,wtf\(\text{values}\)

Good, old, simpler C++03 times...

```cpp
// lvalue
a = 3;
b[4] = 'x';
...
```

```cpp
// rvalue
a = 3;
b[4] = foo();
...
```
Deep Dive: l,r,gl,pr,x,wtfvalues

Now we need something to identify values that can be moved from

---

**lvalue („left value“)**

```
    a = 3;
b[4] = 'x';
... 
```

**xvalue**

```
    std::move(c);
... (not many)
```

**rvalue („right value“)**

```
    a = 3;
b[4] = foo();
... 
```
Deep Dive: l,r,gl,pr,x,wtfvalues

```
// lvalue
a = 3;
b[4] = 'x';
...

// xvalue
std::move(c);
... (not many)

// prvalue
a = 3;
b[4] = foo();
...```

Hasso Plattner Institut
Ensure Moves

```cpp
string(const string& that) {
  size_t size = strlen(that.buf) + 1;
  char* buf = malloc(size);
  memcpy(buf, that.buf, size);
}
string(const string& that) = delete;
```
What does this mean for Opossum?

- You hopefully now have a better idea why we delete the copy constructors and how moves work.

```cpp
void Table::append(std::initializer_list<AllTypeVariant> values) {
  if (_chunk_size > 0 && _chunks.back().size() == _chunk_size) {
    Chunk newChunk;
    for (auto &&type : _column_types) {
      newChunk.add_column(make_shared_by_column_type<BaseColumn, ValueColumn>(type));
    }
    _chunks.push_back(std::move(newChunk));
  }
  _chunks.back().append(values);
}
```
Named Return Value Optimization

```cpp
string get_foo() {
    return string("foo");
}

string get_baz() {
    return std::move(string("baz"));
}

int main() {
    get_foo();
    get_baz();
}
```

```
[~:/Desktop/tmp] $ g++-6 test.cpp -std=c++1z -Wall -Wextra -O3 && ./a.out
allocated 4 bytes for "foo" (constructor)
allocated 4 bytes for "baz" (constructor)
moved baz
```

Next Steps

• Please remember to submit your code reviews for sprint 1
• Dictionary Compression by 16 Nov
• Any Questions about Sprint 2?
ILIW