# Выбрасываем Java и кратно ускоряем Spark/Presto… Или пока нет?

Павел Солодовников, Querify Labs

Email: psolodovnikov@querifylabs.com

# О чём будет речь?

1. Концепция Deconstructed Database
    a. Чем плохи монолитные системы?
    b. Чем хороши модульные системы?
    c. Краткая история и эволюция
    d. Почему сейчас?
2. Тренд на Native Execution — Кейс компании Meta*
3. Чем хорошая векторизация вычислений/данных?
4. Как Meta* решила свои проблемы? Velox

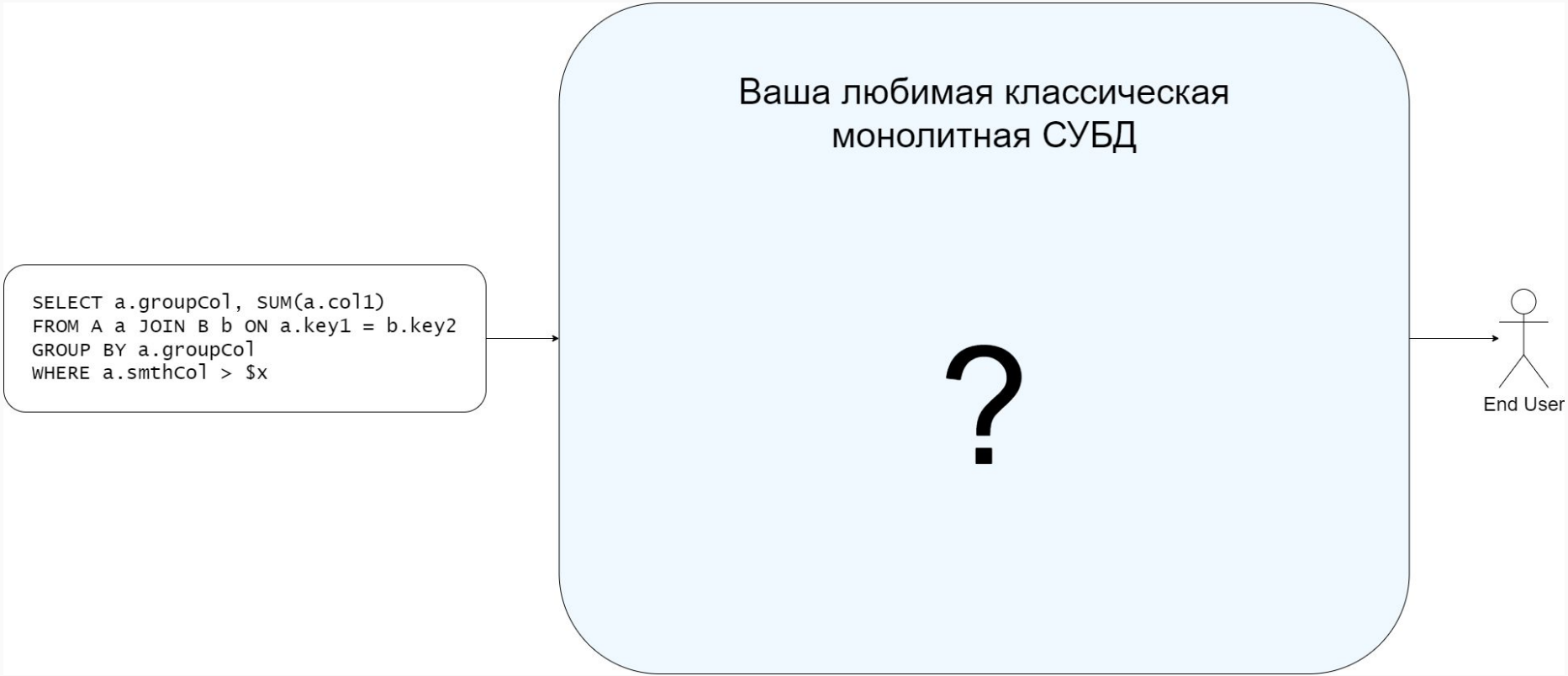
* Входит в перечень общественных объединений и религиозных организаций, в отношении которых судом принято вступившее в законную силу решение о ликвидации или запрете деятельности по основаниям, предусмотренным Федеральным законом от 25.07.2002 № 114-ФЗ «О противодействии экстремистской деятельности»

# О чём будет речь?

5. Основные фичи Velox
6. Существующие интеграции Velox с аналитическими системами
    a. Prestissimo
    b. Apache Gluten
    c. PyVelox
    d. PyTorch/TorchArrow
7. Summary — ключевые моменты
8. Полезные ссылки / Как попробовать?

# Концепция Deconstructed Database

# Запрос к традиционной монолитной СУБД



# Чем плох этот подход?

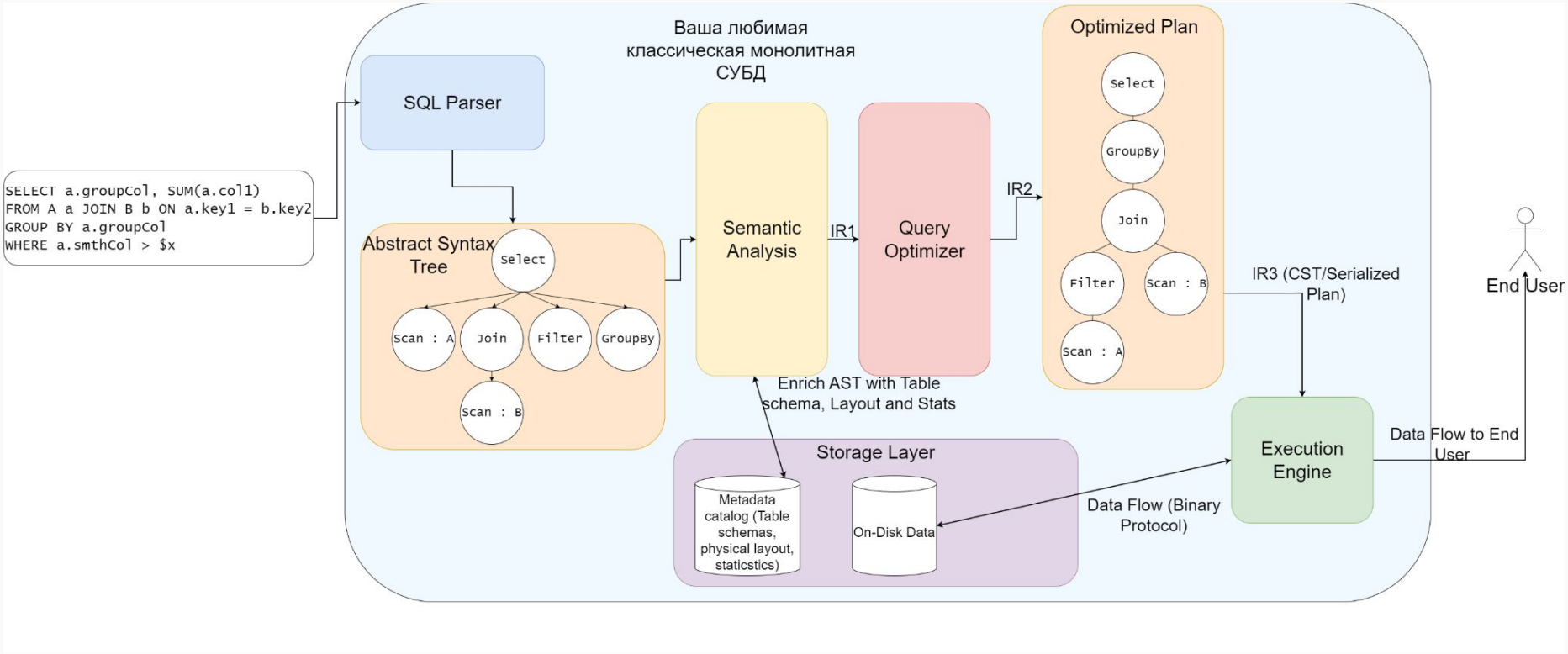
1. “Чёрный ящик” для администратора - ограниченные наблюдение и контроль за поведением системы
2. Сложно масштабировать
3. Lock-in на конкретный диалект SQL, протокол общения с БД (чаще всего кастомный), клиентские библиотеки, заточенные под одну СУБД
4. Ограниченная кастомизация

# Что внутри СУБД?

Так или иначе, любая СУБД состоит из логических частей, таких как:

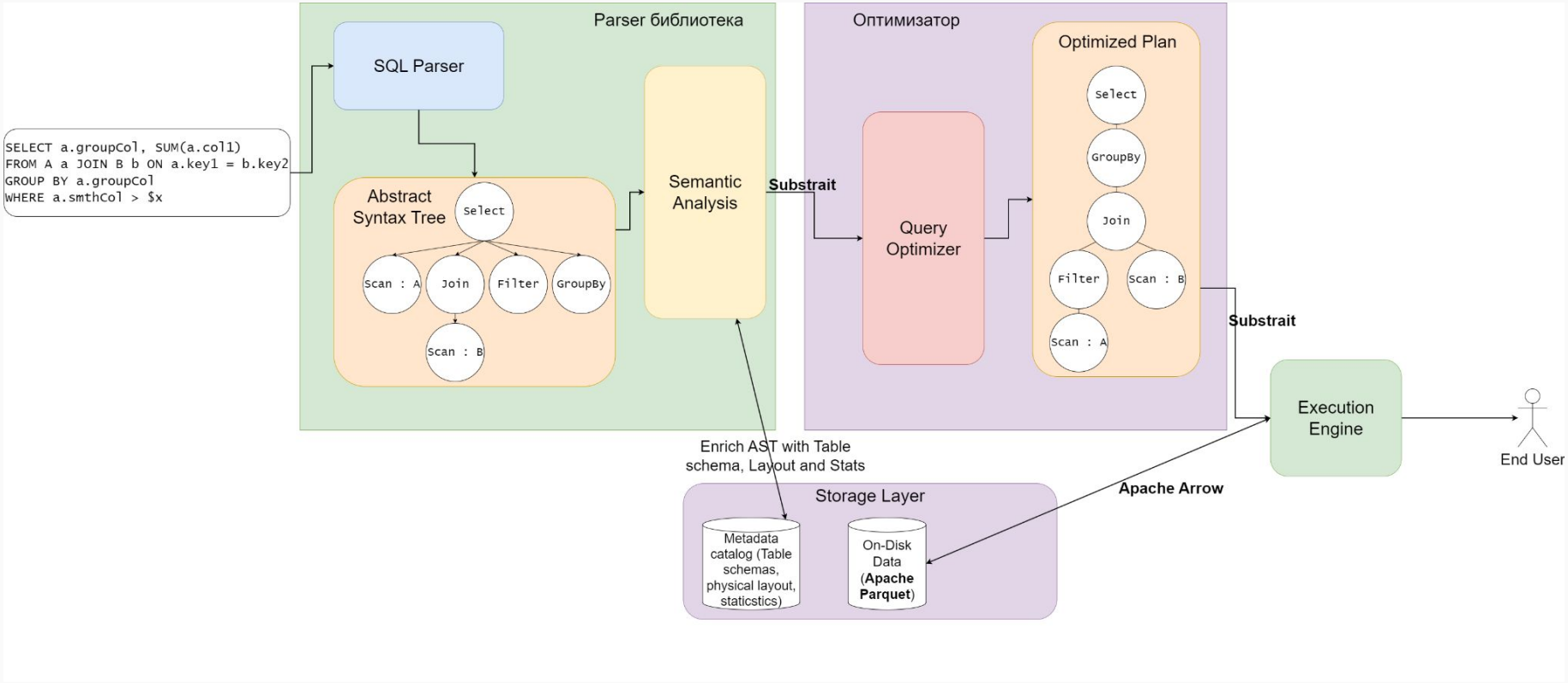
1. Парсер — например, парсинг SQL кода
2. Семантический анализ — обогащение AST информацией о таблицах, колонках и проч.
3. Оптимизатор запросов
4. Слой Storage и Persistency — обращение к хранилищу данных, логу транзакций
5. Слой исполнения запросов — Execution Layer

# Запрос к традиционной монолитной СУБД - Что внутри?



# А что если разбить всё на компоненты и использовать открытые стандарты?

# Модульная СУБД



# Преимущества модульной структуры СУБД

1. Больше контроля над системой и её компонентами
2. Легче интегрировать в свое решение благодаря открытым стандартам — предсказуемое единообразное поведение
3. Больше возможностей кастомизации
    a. Пайплайнинг нескольких оптимизаторов, пост-процессинг
    b. Поддержка новых источников и форматов данных
    c. Добавление недостающих функций (UDFs, UDAFs)
    **d. Недостаточная производительность Execution Engine? Можно заменить на другой!**

# Deconstructed Database — Краткая история и эволюция

- Hadoop и MapReduce (2006)
- Apache Hive — SQL поверх Hadoop (2011)
- Apache Parquet — формат хранения колоночных данных (2013)
- Apache Calcite — SQL парсер+оптимизатор в виде библиотеки (2014)
- Apache Arrow — открытый фреймворк для работы с колоночными данными (2016)
- Apache DataFusion — открытый фреймворк для парсинга, оптимизации, а также среда исполнения запросов (2017)
- Apache Iceberg — открытый формат хранения аналитических данных (2017)
- DuckDB — открытая In-process СУБД, аналог SQLite для аналитических нагрузок (2019)
- **Velox** — фреймворк для реализации векторизованных движков исполнения запросов (2020)

# Deconstructed Database — Почему сейчас?

- Создавать новые СУБД «с нуля» сложно и слишком «дорого»
- Реализация новых, сложных сервисов обработки данных через композицию существующих решений — быстрее Time To Market
- Обилие разнообразного «железа»: общие (CPU) и узкоспециализированные решения (GPU, TPU, NPU, FPGA, ASIC)
- Различные модели вычислений, например: OLAP vs. OLTP, local vs distributed
- SQL vs NoSQL/Non-SQL API: например, появление DataFrame APIs в Pandas, Polars, DataFusion и проч.
- ML / Data Science — возможность “крутить” данные множеством способов: Pre- и Post-Processing, извлечение Бизнес-аналитики

# Тренд на Native Execution — Кейс компании Meta*

* Входит в перечень общественных объединений и религиозных организаций, в отношении которых судом принято вступившее в законную силу решение о ликвидации или запрете деятельности по основаниям, предусмотренным Федеральным законом от 25.07.2002 № 114-ФЗ «О противодействии экстремистской деятельности»

# Problem Statement: Различные рабочие нагрузки

- Внутренняя инфраструктура требует работы со многими источниками данных
- Паттерны использования разнообразные: OLAP и OLTP; стриминг данных и Data Ingestion; ML и AI задачи
- Interactive vs. Batched OLAP Workloads
- Системы эволюционируют отдельно друг от друга → накапливаются мелкие различия (например, поведение аналитических функций) → сложно поддерживать

# Problem Statement: Ограниченная масштабируемость

- Объёмы данных исчисляются терабайтах/петабайтах
- Типичные кластеры для Presto/Spark исчисляются сотнями узлов
- Java воркеры недостаточно хорошо масштабируются с приростом мощностей железа на каждый конкретный инстанс
- Приходится масштабировать горизонтально → большие накладные расходы на содержание кластеров
- Высокие Tail Latencies благодаря GC (Garbage Collection)

# Proposed Solution — Нативное исполнение

Решено было отказаться от Java в Presto/Spark воркерах, что решило сразу несколько проблем:

- Нет JVM и GC → Нет “Stop The World” и прочих накладных расходов, связанных с JVM
- Векторизация вычислений и векторное исполнение выражений (Expressions)
- Переход на Arrow-совместимый формат → легче и дешевле передавать данные между разными системами
- Ближе к железу, больше возможностей использовать SIMD (SSE2, AVX2, AVX512)
- Абстрагирование мелких отличий между разными движками (Presto и Spark)
- Расширяемость и переиспользование кода для разных аналитических систем

# Чем хороша векторизация вычислений/данных?

# Постановка кейса

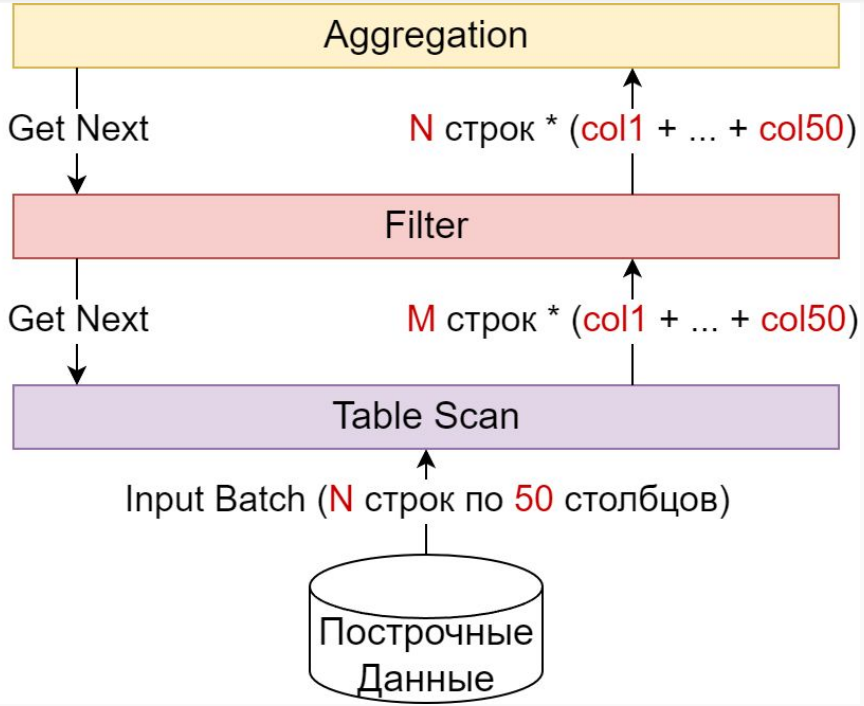
- Есть таблица `T` с 50 колонками (`col1`, `col2`, …, `col50`)
- Есть запрос с агрегацией, использующий всего 2 колонки из этой таблицы:

```
SELECT max(col1) FROM T WHERE col50 > 100
```

# Построчное (Row-major) исполнение

Запрос

```
SELECT max(col1) FROM T
WHERE col50 > 100
```



# Построчное исполнение: Недостатки

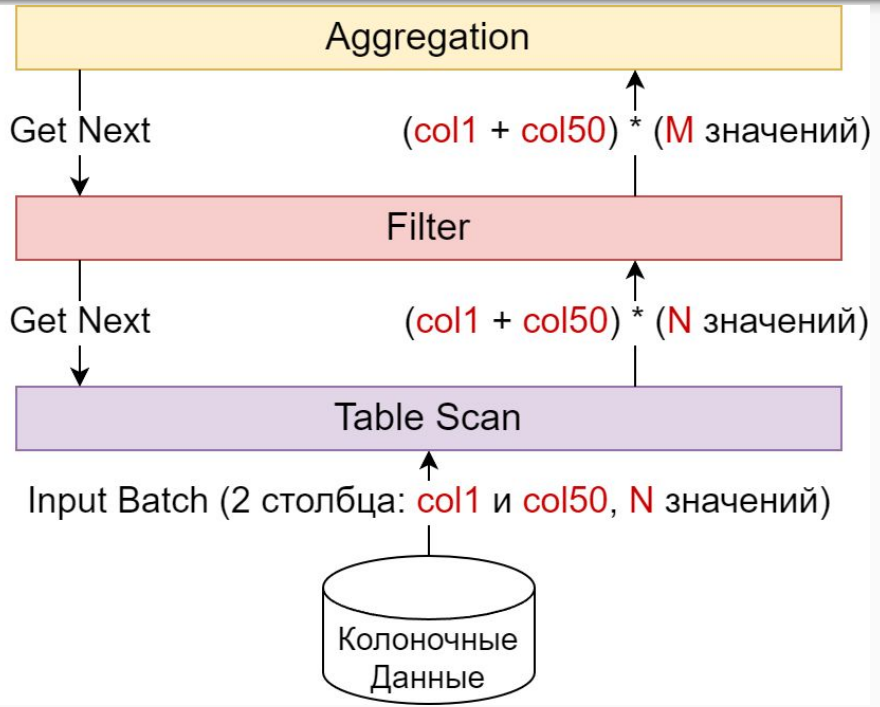
1. Не получится отсекать ненужные колонки на этапе Table Scan
2. Приходится передавать все столбцы (50!) на каждом шаге
3. Значительно возрастает количество данных, которые нужно обрабатывать
4. Строчные данные могут перестать влезать в L1/L2 кеши CPU, если колонок будет слишком много и/или значения очень большие
5. Дольше вычисления, больше затраты по памяти

**Вывод: Для OLAP нагрузок недееспособное решение!**

# Колоночное (векторизованное) исполнение

Запрос

```
SELECT max(col1) FROM T
WHERE col50 > 100
```



# Векторизованное исполнение: Преимущества

1. Легко читать только те колонки с диска, которые нужны (column pruning), если данные в колоночном формате (Parquet, ORC)
2. Нет лишнего Data movement между физическими операторами
3. Всегда одинаковые по ширине данные
4. Больше данных помещается в кеши CPU
5. Легче применять SIMD и параллелизовать вычисления
6. Материализация данных непосредственно перед отдачей пользователю (например, в рамках финального Project)
7. Множество эффективных вариантов для поколоночного сжатия данных (RLE, Delta encoding…)

**Вывод: Для OLAP векторизация — Is A Must!**

# Как Meta* решила свои проблемы?

* Входит в перечень общественных объединений и религиозных организаций, в отношении которых судом принято вступившее в законную силу решение о ликвидации или запрете деятельности по основаниям, предусмотренным Федеральным законом от 25.07.2002 № 114-ФЗ «О противодействии экстремистской деятельности»

# Velox — Что в итоге получилось?

- Библиотека на C++, а не Standalone продукт
- Требует ручной интеграции
- Не включает в себя парсер запросов и оптимизатор → ожидает оптимизированный план на входе
- Реализует слой Execution Engine
- Движок может быть легко встроен как в Presto, так и в Spark, так и в любую другую систему

# Proposed Solution — Disclaimer

Ни в статье про Velox из VLDB'22, ни где-либо ещё не удалось найти детального анализа причин, почему же на самом деле тормозит Java воркер на примере Presto, например:

- Нет данных профилирования (`perf`)
- Нет перечисления конкретных причин, выявленных в результате глубокого анализа имплементации на Java

Поэтому далее поверим коллегам из Meta* наслово, что они этот анализ всё-таки провели :)

# Velox — One Execution Engine To Rule Them All

- Есть много систем и каждая из них имеет свою среду выполнения (Execution Engine)
- У каждой — свои особенности, нюансы (quirks), а также ограничения по производительности (чаще всего, исторические)

# Velox — One Execution Engine To Rule Them All

- Заменяем движок исполнения на Velox → получаем единообразное поведение во всех системах
- Доработки под каждую из систем строго изолированы внутри расширений Velox
- Оптимизируем движок — прибавку в производительности получают сразу все

# Velox — основные фичи: Векторизация

- **Векторизованные вычисления**
- **Оптимизация и компиляция выражений**
- **Поддержка Zero-copy для Arrow данных**

# Velox — основные фичи: Управление памятью

- Векторизованные вычисления
- Оптимизация и компиляция выражений
- Поддержка Zero-copy для Arrow данных
- **Настраиваемые Memory Pool’ы**
- **Memory Arbitration и Spilling - автоматический реклейминг памяти и сбрасывание данных на диск в условиях нехватки ресурсов**
- **Оптимизированные Hash Joins и операции со строками**
- **Локальный LRU-кеш для данных на SSD**

# Velox — основные фичи: Расширяемость

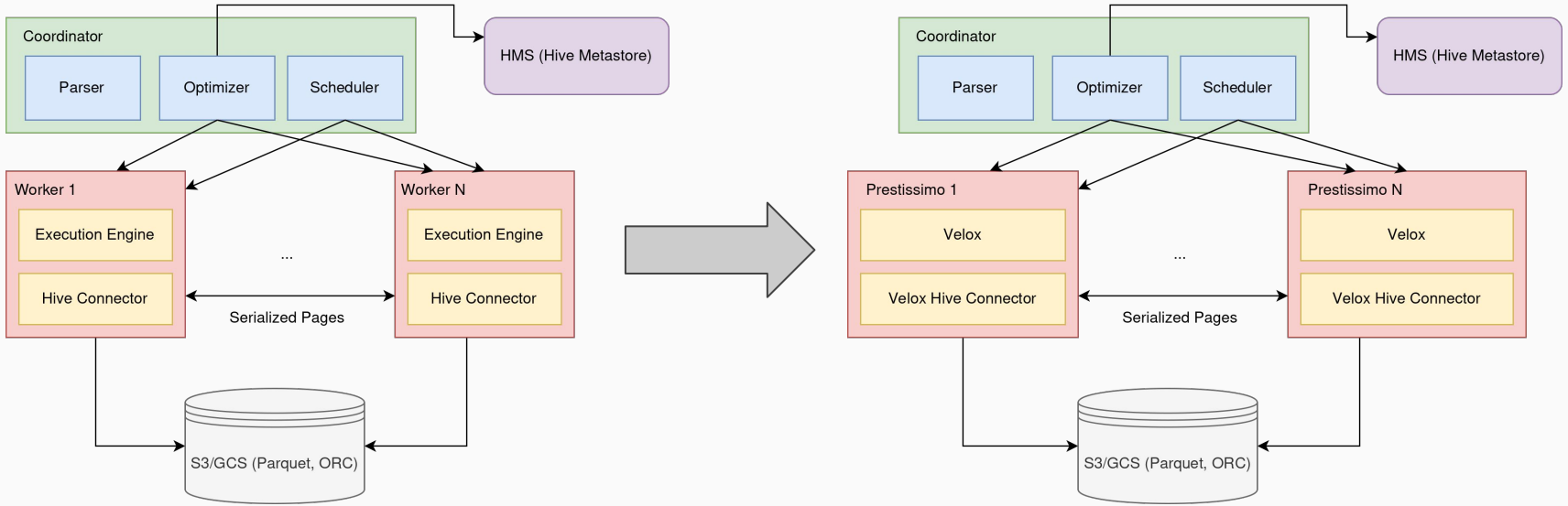
- Векторизованные вычисления
- Оптимизация и компиляция выражений
- Поддержка Zero-copy для Arrow данных
- Развитое управление памятью через Memory Pool’ы
- Memory Arbitration и Spilling - автоматический реклейминг памяти и сбрасывание данных на диск в условиях нехватки ресурсов
- Оптимизированные Hash Joins и операции со строками
- Локальный LRU-кеш для данных на SSD
- **Поддержка Apache Substrait**
- **Поддержка большого количества Presto и Spark функций**
- **Интерфейс коннекторов к удаленным источникам данных**
- **Поддержка Parquet, ORC форматов хранения данных**
- **Расширяемость**

# Velox — Существующие интеграции

# Velox — текущие интеграции

- Prestissimo (или Presto Native Execution) — C++ воркер для Presto
  - **(Более-менее Production Ready)**
- Apache Gluten — C++/Java плагин для Spark воркеров
  - **(Highly Experimental!)**
- PyVelox — Python обвязка для C++ API библиотеки
  - **(Experimental, но на “поиграться” можно использовать)**
- PyTorch / TorchArrow — доступ к Presto функциям из Velox через `from torcharrow import functional`
  - **(Уже в Upstream, можно использовать!)**

# Velox - Presto интеграция: Prestissimo



# Velox - Prestissimo: Бенчмарки

Относительный прирост в скорости обработки разнообразных запросов на реальных аналитических ворклоадах внутри Meta*:

**От 1.5x до 10x**

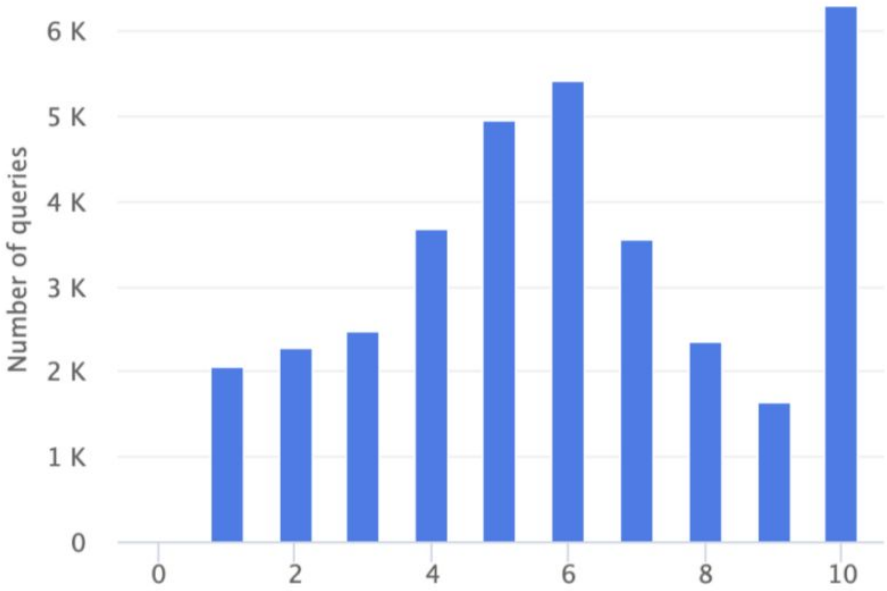



**Figure 4: Prestissimo speedup over Presto Java under real interactive analytical workloads. The bars represent how many times Prestissimo is faster than Presto Java.**

* Входит в перечень общественных объединений и религиозных организаций, в отношении которых судом принято вступившее в законную силу решение о ликвидации или запрете деятельности по основаниям, предусмотренным Федеральным законом от 25.07.2002 № 114-ФЗ «О противодействии экстремистской деятельности»

Источник: https://vldb.org/pvldb/vol15/p3372-pedreira.pdf

# Velox - Prestissimo: Бенчмарки (TPC-H)

Датасет: TPC-H 3TB

Кластер из 80 Presto узлов (воркеров)

64G RAM и 2x2TB SSD

ORC формат без zstd сжатия

Velox кластер показывает сравнимую производительность с Java кластером требуя в **3x** меньше узлов!

**Table 2: TPC-H results comparing Prestissimo (Velox's C++ engine) vs. Presto Java engine.**

| | Wall time (sec) | | | CPU time (sec) | | |
|---|---|---|---|---|---|---|
| | C++ | Java | Speedup | C++ | Java | Speedup |
| Q1 | 5 | 42 | 8.4x | 2211 | 14435 | 6.5x |
| Q6 | 1 | 9 | 9x | 538 | 2018 | 3.7x |
| Q13 | 15 | 31 | 2x | 5647 | 12322 | 2.1x |
| Q19 | 6 | 13 | 2.1x | 1362 | 3483 | 2.5x |

Источник: https://vldb.org/pvldb/vol15/p3372-pedreira.pdf

# А что с Trino?

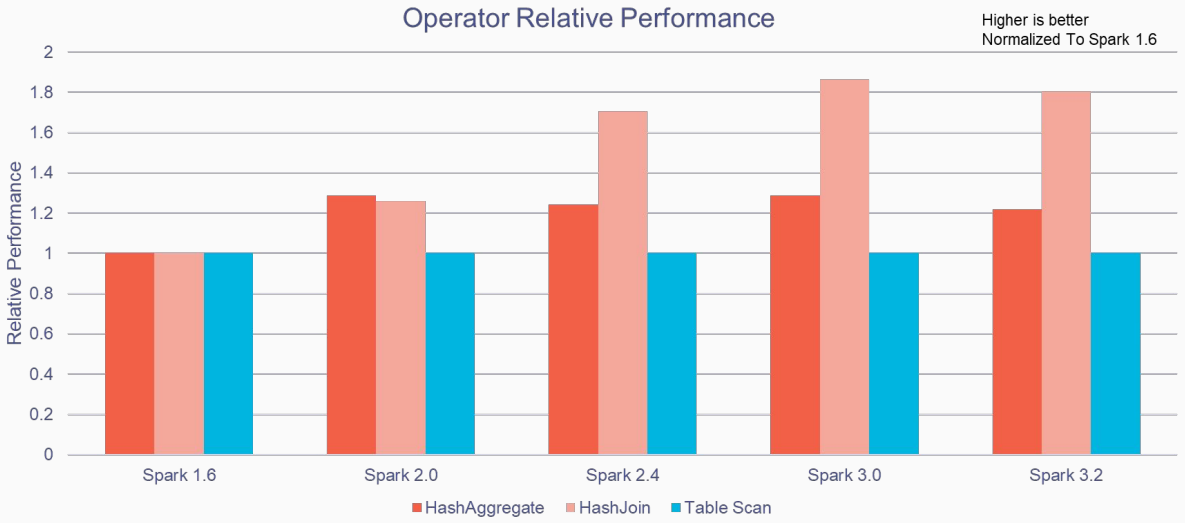
К сожалению, Prestissimo просто так прикрутить к Trino не получится :(

Есть проект Gluten-Trino, он пошел по другому пути, нежели Prestissimo, но главное, что проект более не развивается.

Репозиторий помещён в архив, но для особо заинтересовавшихся его всё ещё можно посмотреть и даже попробовать поиграться: https://github.com/oap-project/Gluten-Trino

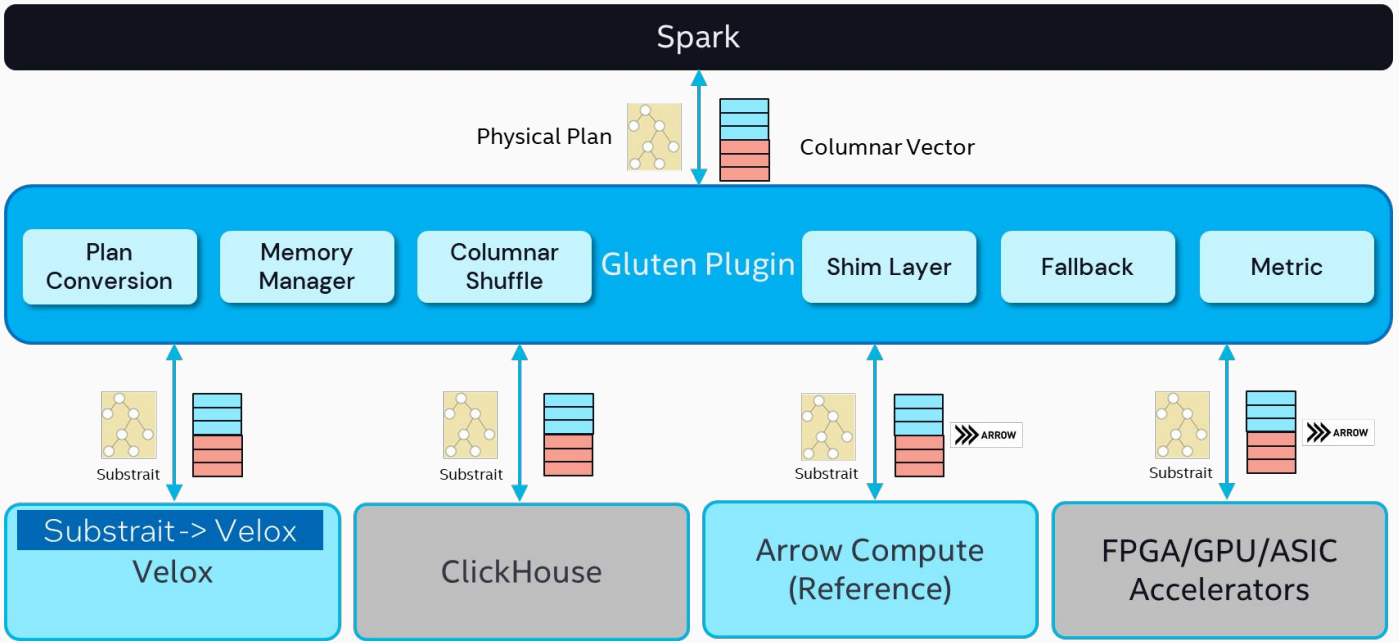
# Velox - Gluten интеграция: Мотивация

- Производительность Java воркеров стоит на месте и не улучшается с новыми релизами
- Только горизонтальное масштабирование
- Вместо этого пытаемся улучшить ядро исполнения каждого воркера, используя новейшие технологии




Источник: https://gluten.apache.org/

# Velox - Gluten интеграция: Архитектура




Источник: https://gluten.apache.org/

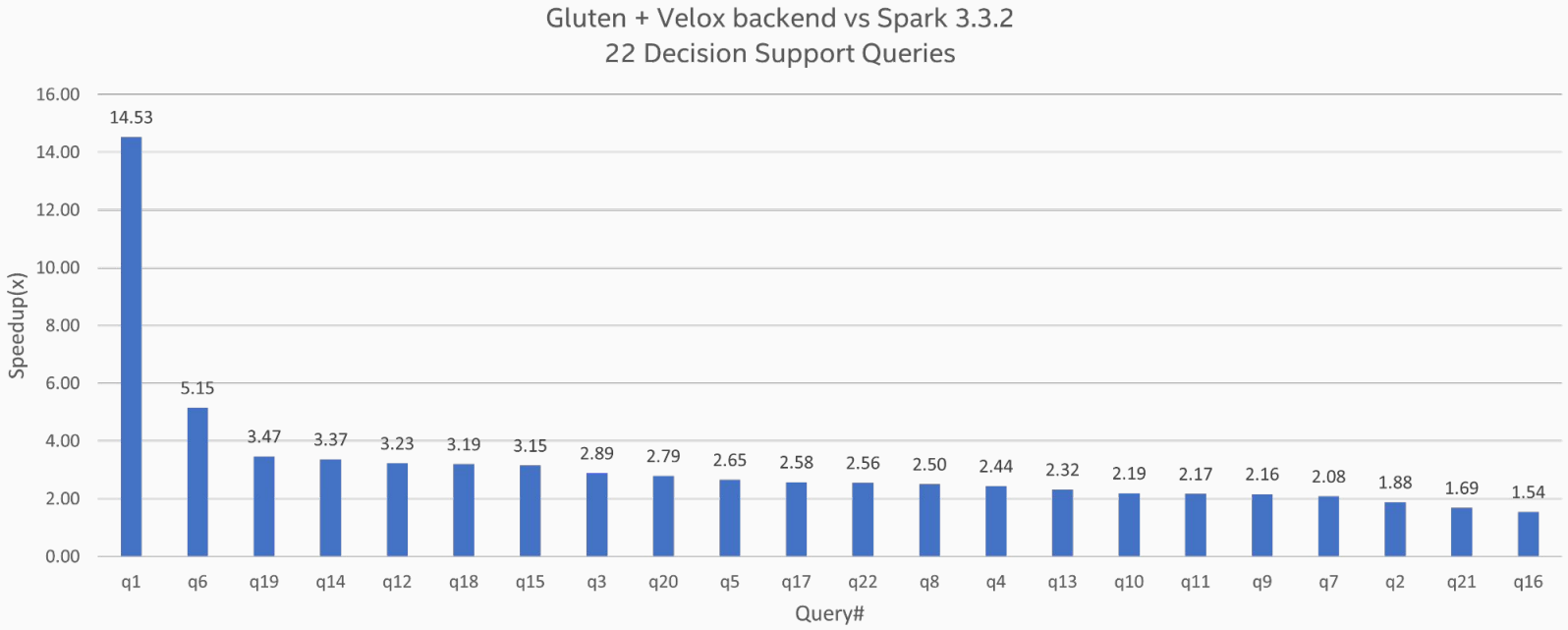
# Apache Substrait — Открытый стандарт представления планов реляционной алгебры

- Кросс-языковой формат сериализации планов исполнения для реляционных СУБД
- Основан на Protobuf
- Открытый стандарт взаимодействия между оптимизаторами, планировщиками и движками исполнения запросов

# Velox - Gluten интеграция: Бенчмарк TPC-H

- 2TB данных
- Spark 3.3.2 в качестве Baseline и для Gluten
- Среднее ускорение запросов в **2.71x** усреднённо по всем запросам
- Вплоть до **14.53x** ускорения на индивидуальном запросе (Q1)
- **Проект ещё нестабилен и активно развивается, результаты бенчмарков будут меняться!**

# Velox - Gluten интеграция: Бенчмарк TPC-H




Источник: https://github.com/apache/incubator-gluten/blob/main/README.md

# Velox - PyVelox интеграция: Python обвязка для C++ API

- Повторяет API C++ библиотеки, но в Питоне — достаточно низкоуровневая библиотека
- Статья с индийского PyCon'23 про PyVelox: https://in.pycon.org/cfp/pycon-india-2023/proposals/pyvelox-interfacing-python-bindings-for-the-unified-execution-engine-by-meta~eXDo5/
- Есть пакет `PyVelox` на PyPi: https://pypi.org/project/pyvelox/ — но староват, лучше собирать самостоятельно из исходников

# Velox - PyTorch интеграция: TorchArrow

- Уже доступно через `from torcharrow import functional`: https://pytorch.org/torcharrow/beta/functional.html
- Все основные Presto функции доступны (полный список можно посмотреть здесь: https://facebookincubator.github.io/velox/functions.html)

# Summary — Ключевые моменты

- Velox — многообещающий инструмент, способный значительно ускорять OLAP запросы (от 2x до >10x на TPC-H, например)
- Меньше затраты CPU Time → меньше требуется серверов → меньше $$$ на обслуживание кластера!
- Есть зрелые интеграции с Presto через Prestissimo, а также с PyTorch
- Можно попробовать поиграться с Apache Gluten (но до прода ещё далеко!)
- В будущем будем видеть намного больше примеров построения систем на основе Decomposed Database (Datafusion, DuckDB и проч.)

# Спасибо за внимание!

Павел Солодовников, Querify Labs

Email: psolodovnikov@querifylabs.com

# Полезные ссылки “на почитать”/поиграться: Velox

- VLDB’22 Paper: https://vldb.org/pvldb/vol15/p3372-pedreira.pdf
- Официальная документация: https://facebookincubator.github.io/velox/

  Много технических деталей, есть примеры использования C++ API

- Доки по сборке и развертыванию (`docker-compose`) Prestissimo воркеров: https://github.com/prestodb/presto/tree/master/presto-native-execution

# Полезные ссылки “на почитать”/поиграться: Apache Gluten

- Официальная документация: https://gluten.apache.org/

  Много полезных сведений о том, как это работает

- Getting Started вместе с Velox бэкендом: https://gluten.apache.org/docs/getting-started/velox-backend/

# Полезные ссылки “на почитать”/поиграться: PyVelox

- Официальная страница пакета на PyPi: https://pypi.org/project/pyvelox/
- Лучше собирать из исходников — на PyPi лежит очень старый пакет!
- Низкоуровневые bindings к C++ библиотеке
- Есть хорошая статья с индийского PyCon про PyVelox вместе с примерами: https://in.pycon.org/cfp/pycon-india-2023/proposals/pyvelox-interfacing-python-bindings-for-the-unified-execution-engine-by-meta~eXDo5/

# Полезные ссылки “на почитать”/поиграться: PyTorch/TorchArrow

- Официальная документация по всем поддерживаемым функциям с примерами: https://pytorch.org/torcharrow/beta/functional.html
- Довольно большое покрытие Presto функций на текущий момент
- Уже довольно давно в Upstream, можно использовать