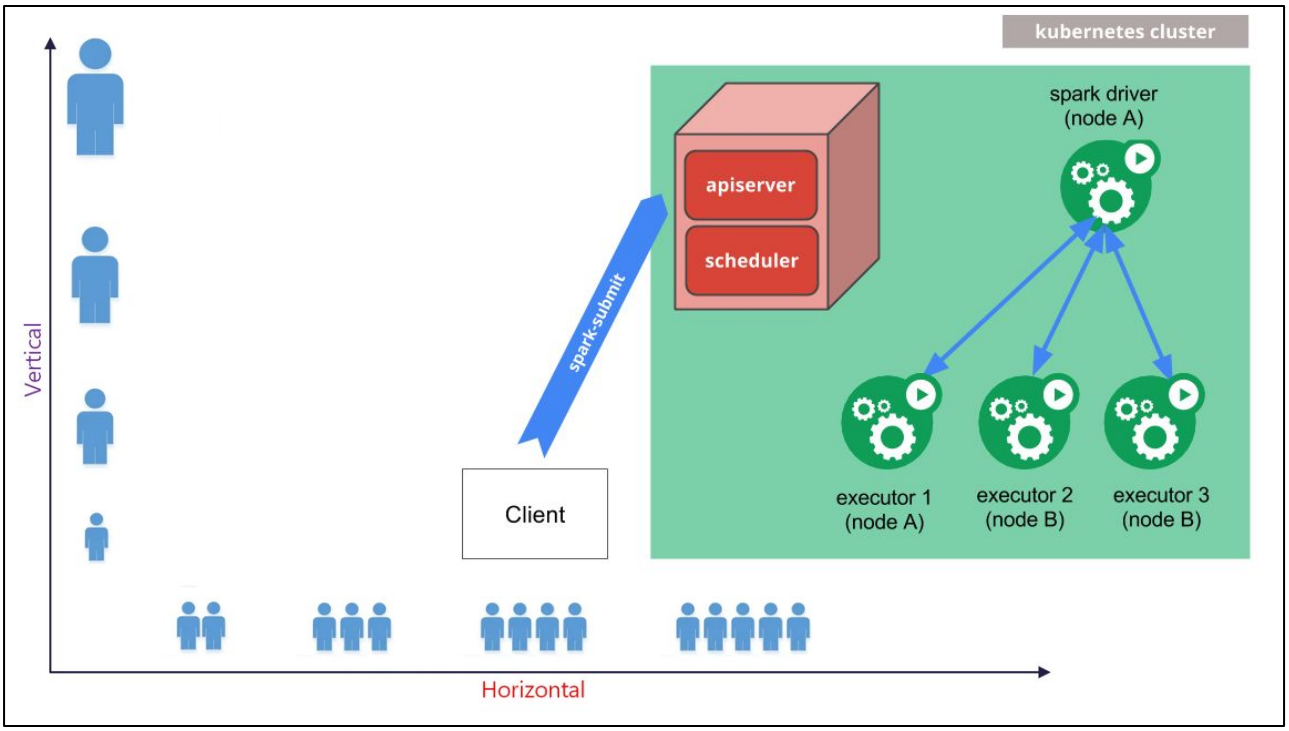
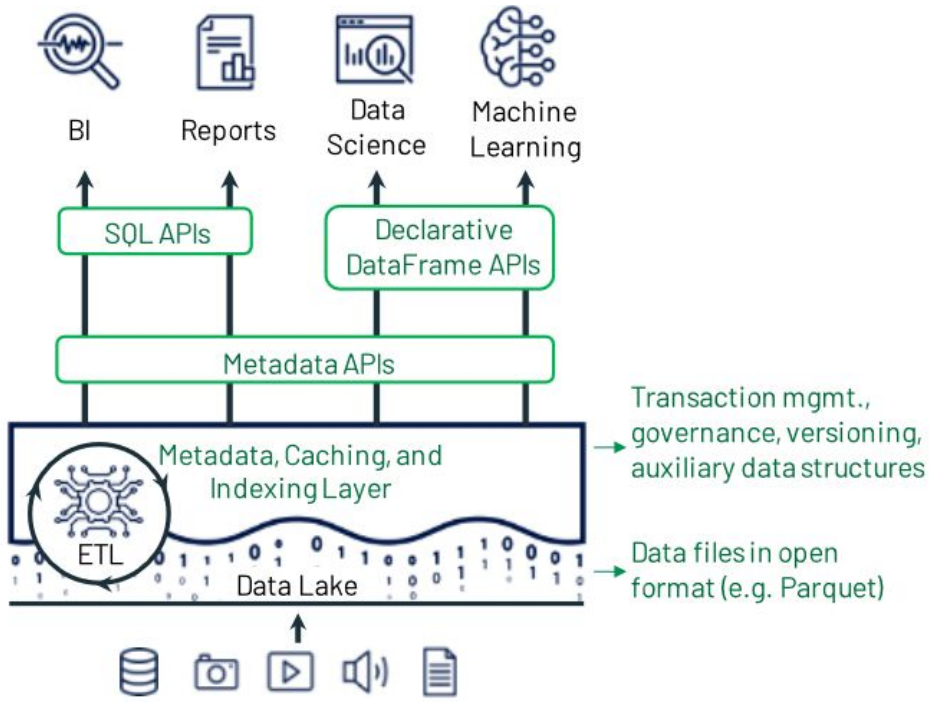
Инструментарий Delta Lake и архитектура Lakehouse



Хранилища данных имеют долгую историю применения в приложениях поддержки принятия решений и бизнес-аналитики. С момента своего создания в конце 1980-х годов технология хранилищ данных продолжала развиваться, сначала через вертикальное масштабирование, а затем, с развитием сетевых технологий, и горизонтального.

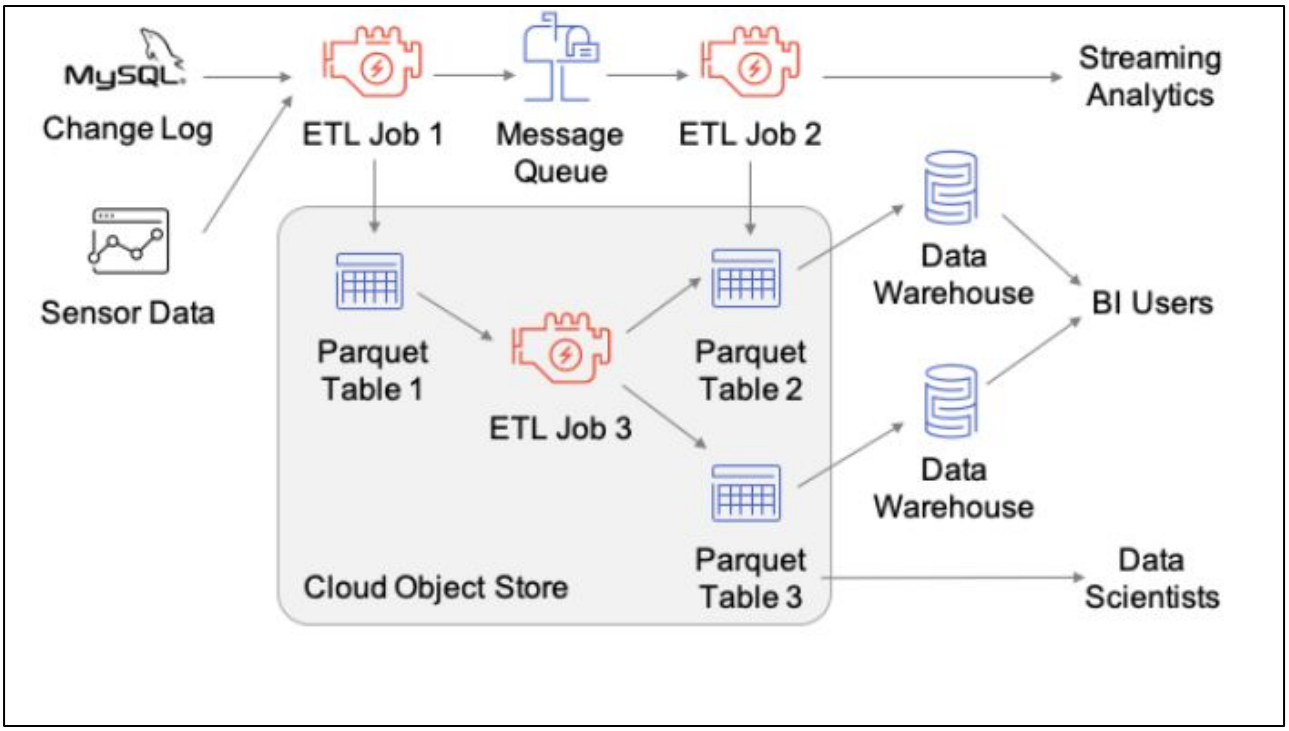
В итоге, архитектура MPP привела к созданию систем, способных обрабатывать данные большего размера. На сегодняшней день венцом эволюции систем обработки больших данных является Spark, а с внедрением в него Delta Lake получила распространение так называемая архитектура “Дома на озере” или Lakehouse.

Пример архитектуры “Дома на озере”



Хотя хранилища отлично подходят для хранения структурированных данных, многим организациям приходится иметь дело с частично- / не-структурированными данными. Когда компании сфокусировались на обработке больших объемов данных из разных источников, архитекторы начали задумываться о единой системе хранения данных для различных видов расчетов таких как ML, BI и пр.

Архитектура Lakehouse сочетает в себе стандартные функции управления СУБД, заменяя при этом отдельное озеро и хранилище данных, а также системы NRT стриминга. Рассмотрим пейпер Databricks – автора Delta Lake.

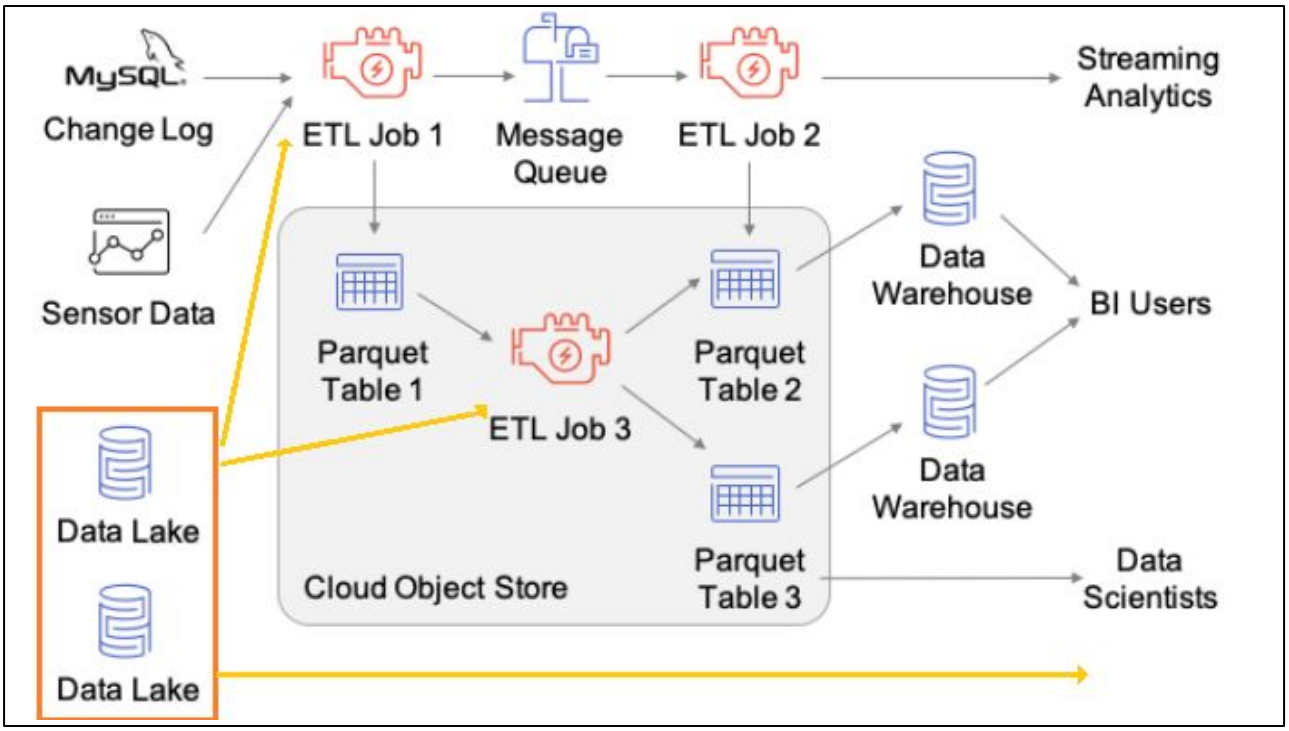


Около десяти лет назад компании начали создавать озера данных – хранилища необработанных данных в различных форматах. Хотя озера данных подходят для хранения различных данных, им не хватает некоторых важных функций:

- Не поддерживают транзакции.
- Не обеспечивают качество данных.
- Нет согласованности/изоляции.

Это осложняет смешивание операций добавления и чтения, а также пакетных и потоковых заданий. По этим причинам многие обещания, связанные с озерами данных, не были имплементированы. Потребность в гибкой, высокопроизводительной системе не уменьшилась. Компаниям все еще требуются

универсальные системы обработки больших данных, включая аналитику на базе QL, а также мониторинг в реальном времени и машинное обучение.



Большинство последних достижений в области искусственного интеллекта связаны с улучшением моделей обработки неструктурированных данных (текста, изображений, видео, аудио), но это именно те типы данных, для которых хранилище данных не оптимизировано.

Тогда компании начали создавать озера данных — хранилища необработанных данных в различных форматах. Хотя озера данных подходят для хранения данных, им не хватает некоторых важных функций:

- не поддерживают транзакции,
- не обеспечивают качество данных,
- нет согласованности/изоляции.

Это осложняет смешивание операций добавления и чтения, а также пакетных и потоковых заданий. По этим причинам многие обещания, связанные с озерами данных, не были реализованы, что во многих случаях привело к потере многих преимуществ хранилищ данных.

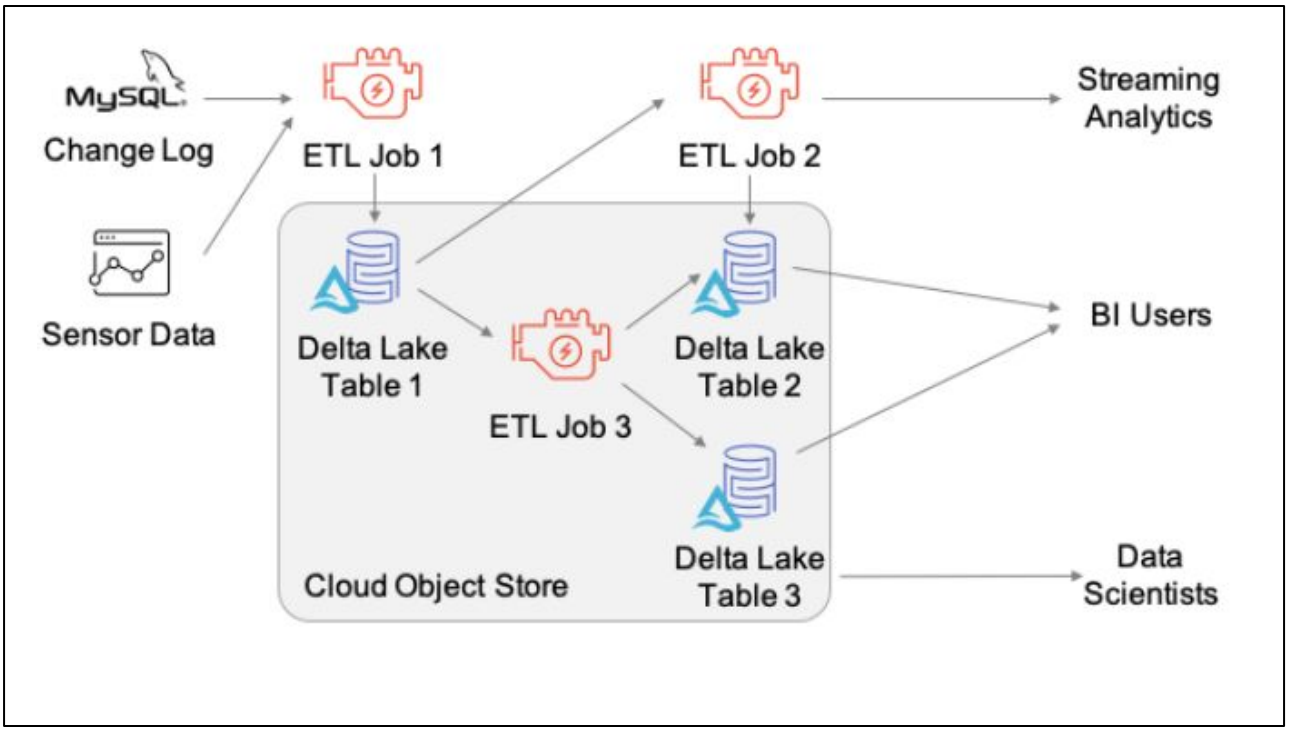
Потребность в гибкой, высокопроизводительной системе не уменьшилась. Компаниям требуются системы для различных приложений обработки данных, включая аналитику на базе QL, мониторинг в реальном времени, обработку данных и машинное обучение.

Поддерживать согласованность озера и хранилища данных сложно и дорого. Требуется непрерывная поддержка пайплайнов между двумя системами и обеспечения их доступности для высокопроизводительной поддержки принятия решений и BI. На каждом этапе также существует риск возникновения сбоев или ошибок, которые снижают качество данных, например, из-за различий между данными в озере и хранилище в моменте до синхронизации пайплайнов. Т.о. данные в хранилище часто устаревают по сравнению с данными в озере данных, а загрузка новых данных часто занимает часы, а то и дни. Это такой шаг назад по сравнению с ранними системами обработки “небольших” данных,

где новые данные были немедленно доступны для запросов, но их было относительно мало.

Согласно опросу Dimensional Research и Five-tran, 86% аналитиков используют устаревшие данные, а 62% сообщают об ожидании инженерных ресурсов несколько раз в месяц. Ограниченная поддержка расширенной аналитики. Несмотря на многочисленные исследования по слиянию машинного обучения и управления данными, ни одна из ведущих систем машинного обучения, таких как TensorFlow, PyTorch и XGBoost не работает хорошо поверх хранилищ. В отличие от BI этим системам необходимо обрабатывать большие объемы данных с использованием сложного кода, отличного от QL.

В качестве альтернативы пользователи могут запускать эти системы на основе данных озера данных в открытых форматах. Однако при этом они теряют богатые функции управления из хранилищ данных, такие как транзакции ACID, управление версиями и индексирование данных.

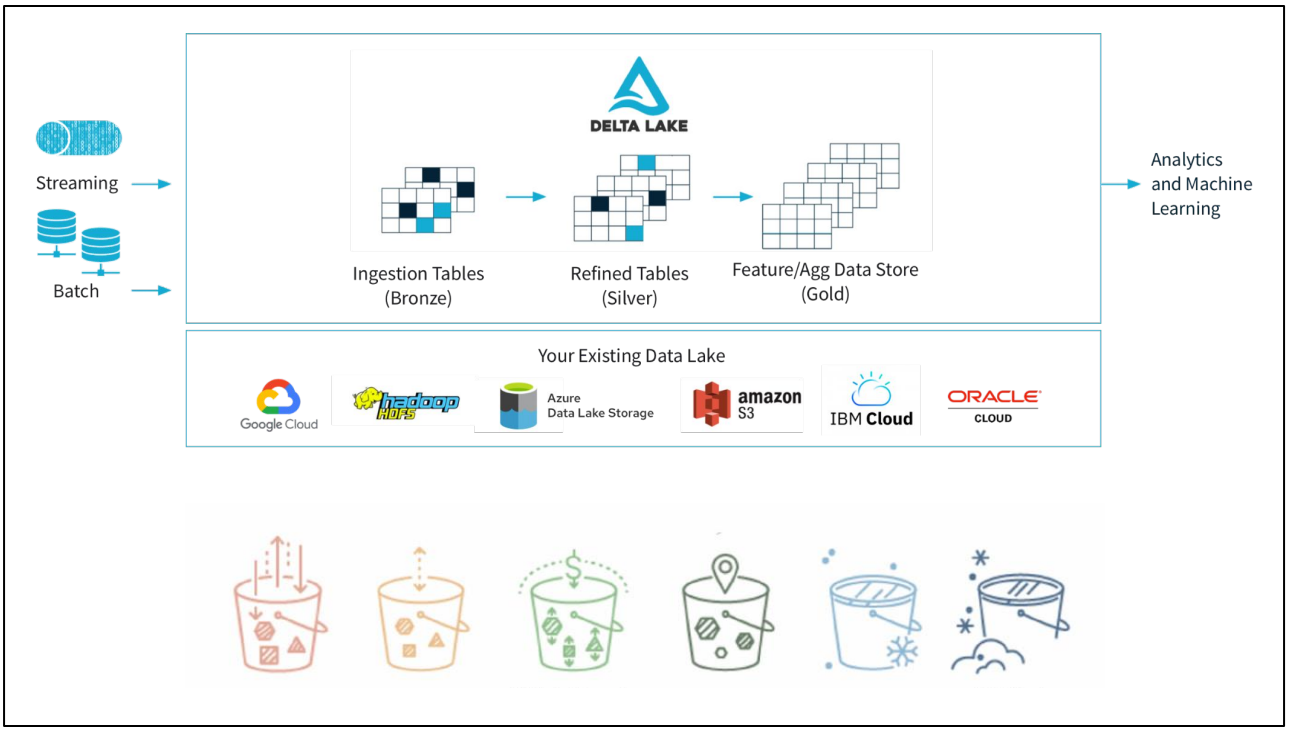


В итоге пользователи платят двойную стоимость хранения данных, скопированных в хранилище, а коммерческие хранилища фиксируют данные в собственных форматах, что увеличивает стоимость миграции данных или рабочих нагрузок в другие системы. Наличие множества систем усложняет работу и, что более важно, приводит к задержкам, поскольку специалистам по обработке данных неизменно приходится перемещать или копировать данные между различными системами.

Delta Lake – это ACID уровень хранения таблиц поверх облачных хранилищ объектов, который DataBricks начали предоставлять клиентам в 2017 году и с открытым исходным кодом в 2019 году. В основе Delta

Lake лежат дельта-таблицы, которые и обеспечивают соответствующие функции для всех этих случаев хранения. Delta Lake устраняет необходимость управлять несколькими копиями данных, сохраняя все в единый Cloud Object Storage.

В Cloud Object Store хранится и информация о том, какие объекты являются частью дельта-таблицы с использованием журнала, который сам хранится там же. Такое решение позволяет клиентам параллельно обновлять несколько объектов, сохраняя скорость обработки. Журнал также содержит метаданные для каждого объекта-файла, оптимизируя поиск для озера данных. Важно отметить, что в Delta Lake транзакции выполняются с использованием оптимистичных блокировок прямо на Cloud Object Store. Это означает, что никакие дополнительные сервисы больше не нужны чтобы поддержать состояние дельта-таблиц.



Теперь поговорим о том зачем дополнительный слой над Cloud Object Storage.

S3 совместимые Cloud Object Storage такие как Amazon S3 и Azure Blob Storage, Google Cloud Storage и OpenStack Swift, предлагают простой и легко масштабируемый интерфейс хранилища объектов. Эти системы позволяют пользователям создавать сегменты / бакеты / ведра, в каждом из которых хранится несколько объектов. Объекты, каждый из которых представляет собой двоичный объект размером до нескольких ТБи.

При чтении объекта облачные хранилища объектов обычно поддерживают байтовое форматирование.

Поэтому эффективно читать блоками (например, байты от 10 000 до 20 000). Это дает возможность использовать форматы хранения, которые группируют часто используемые значения. Обновление объектов обычно требует перезаписи всего объекта за раз. Эти обновления можно сделать атомарными, по аналогии с транзакциями, чтобы видеть либо новую версию объекта, либо старую.

Наиболее популярные Data Cloud Storages обеспечивают eventual consistency для каждого ключа и не гарантируют согласованность между ключами, что создает проблемы при управлении набором данных, состоящим из нескольких объектов, как описано во введении. Пример сходен вопросу про атомики в Java: почему нельзя описать состояние двигателя двумя атомиками если они обеспечивают консистентность в памяти? Да потому что между собой они не консистентны.

В частности, после того, как клиент загружает новый объект, другие клиенты не обязательно сразу же увидят объект в LIST. Аналогично, обновления существующего объекта могут не сразу быть видны другим клиентам. Более того, в зависимости от хранилища объектов даже клиент, выполняющий запись, может не сразу увидеть новые объекты. На это дополнительно накладывается то что точная модель согласованности

различается в зависимости от поставщика облака и может быть довольно сложной.

Конкретный пример: Amazon S3 обеспечивает согласованность чтения после записи для клиентов, записывающих новый объект. Это означает, что операции чтения, такие как GET S3, будут возвращать содержимое объекта после PUT. Однако есть одно исключение: если клиент, записывающий объект, выполнил GET для (несуществующего) ключа перед его PUT, то последующие GET могут не читать объект в течение определенного периода времени, скорее всего, потому, что S3 использует отрицательное кэширование. Более того, операции LIST в S3 всегда в конечном итоге согласованы, а это означает, что LIST после PUT может не вернуть новый объект. Другие хранилища облачных объектов предлагают более строгие гарантии, но им все еще не хватает атомарных операций с несколькими ключами. Это все еще eventual consistency.

Однако ни одно из текущих решений, согласно пейперу Databricks не удовлетворяет архитектуре Lakehouse. Чтобы решить эти проблемы разработан инструмент Delta Lake – уровень хранения таблиц с гарантиями ACID поверх традиционного Cloud Data Storage. Сами объекты данных дельта таблиц закодированы в Parquet, что позволяет легко писать коннекторы для

механизмов, которые уже могут обрабатывать Parquet. Такая конструкция позволяет клиентам обновлять несколько объектов одновременно, заменять подмножество объектов другим, при этом обеспечивая высокую производительность параллельного чтения и записи самих объектов (аналогично необработанному Parquet).

Функции Delta Lake

- Реляционные таблицы с запросами и изменением данных.
- Поддержка транзакций ACID.
- Управление версиями данных и путешествия во времени.
- Поддержка бетч и стрим передачи данных.
- Стандартные форматы и совместимость.
- Внешние или внутренние (управляемые) таблицы.

Рассмотрим основные функции:

1. Реляционные таблицы с запросами и изменением данных. С помощью Delta Lake вы можете хранить данные в таблицах, поддерживающих операции CRUD (создание, чтение, обновление и удаление). Т.о. можно выбирать / вставлять / обновлять / удалять строки данных так же, как и в РСУБД.
2. Поддержка транзакций ACID. РСУБД предназначены для поддержки модификаций транзакционных данных, которые обеспечивают:
    - Атомарность. Транзакции либо применяются, либо откатываются.
    - Согласованность. Транзакции оставляют данные в консистемном состоянии.
    - Изоляция. Транзакции в процессе не мешают

- друг другу.
- Долговечность – транзакция завершается и ее изменения сохраняются даже при рандомном отключении питания.

Delta Lake обеспечивает такую же поддержку транзакций в Spark, вводя журнал транзакций и обеспечивая сериализуемую изоляцию для параллельных операций.

1. Управление версиями данных и путешествия во времени. Поскольку все транзакции регистрируются в журнале транзакций, вы можете отслеживать несколько версий каждой строки таблицы и даже использовать функцию перемещения во времени для получения предыдущей версии строки в запросе.
2. Поддержка бетч и стрим передачи данных. Хотя большинство РСУБД включают таблицы, в которых хранятся статические данные, Spark включает встроенную поддержку потоковой передачи данных через Structured Streaming. Таблицы Delta Lake можно использовать как приемники, так и источники потоковой передачи данных.
3. Стандартные форматы и совместимость. Базовые данные для таблиц Delta Lake хранятся в формате Parquet, который обычно используется в конвейерах приема озера данных. Кроме того, вы можете использовать бессерверный пул QL в

1. Azure Synapse Analytics для запроса таблиц Delta Lake в QL.
2. Таблицы в каталоге Spark, включая таблицы Delta Lake, могут быть управляемыми или внешними. Важно понимать разницу между этими типами таблиц:
    - Управляемая таблица. Определяется без указания местоположения, а файлы данных хранятся в хранилище, используемом хранилищем метаданных. Удаление таблицы не только удаляет ее метаданные из каталога, но также удаляет папку, в которой хранятся ее файлы данных.
    - Внешняя таблица. Определяется для пользовательского местоположения файла, в котором хранятся данные таблицы. Метаданные таблицы определены в каталоге Spark. Удаление таблицы удаляет метаданные из каталога, но не влияет на файлы данных.

Преимущества Delta Lake

- Полноценная работа с историческими данными.
- Операции UPSERT, DELETE и MERGE.
- Потоковый ввод-вывод.
- Кэширование.

- Кластеризация записей данных
- Миграция схемы и версионирование.
- Ведение журнала аудита на основе журнала транзакций.

Рассмотрим преимущества:

1. Полноценная работа с историческими данными на определенный момент времени, которая позволяет откатывать ошибочные обновления своих данных, в отличие от частичного метода на директориях с версиями, который есть в озере данных.
2. Операции UPSERT, DELETE и MERGE, которые позволяют тщательно переписать соответствующие объекты для имплементации обновлений архивированных данных и рабочих процессов обеспечения соответствия (например, GDPR).
3. Потоковый ввод-вывод. В сравнение с Data Lake появляется параллельный ввод-вывод чанками, с

1. координацией объединения в более крупные объекты транзакционно. Это увеличивает производительность. Можно рассматривать дельта-таблицы как шины сообщений для NRT т.к. поддерживаются «хвостовые» чтения новых данных, добавленных в таблицы.
2. Кэширование. В отличие от Data Lake объекты в дельта-таблице и ее журнале являются неизменяемыми, узлы кластера могут безопасно кэшировать их на нодах кластера.

А также:

1. Кластеризация записей данных для локальности без влияния на выполнение запросов. Поддержка Computations Close to Data.
2. Миграция схемы и версионирование для контролируемых изменений и откатов на любую версию без проблем при изменении структуры таблиц.
3. Ведение журнала аудита на основе журнала транзакций, а не директорий с версиями. Любая транзакция, требующая записи / обновления нескольких объектов, исключает частичную запись. Обычно, если пайплайн не отработал, то данные остаются в поврежденном состоянии.

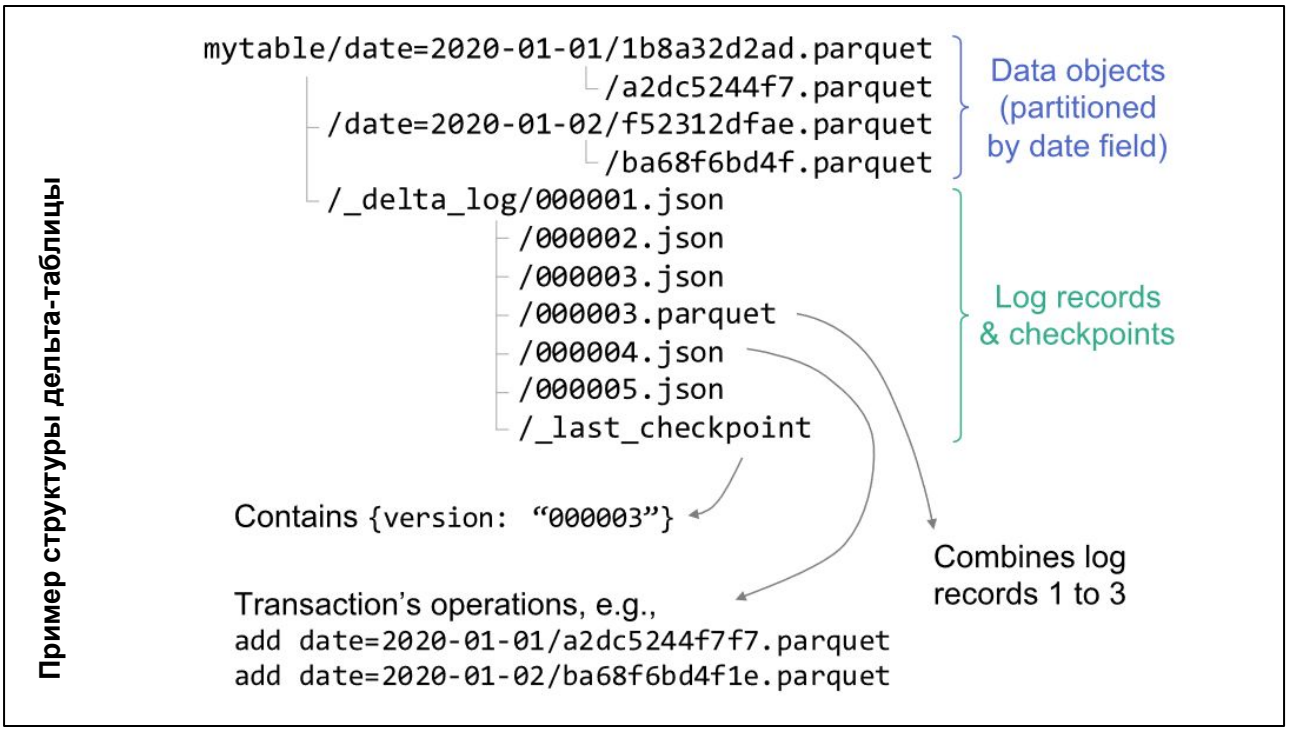
Варианты использования Delta Lake

- Платформы обработки больших данных для ETL /ELT.
- Эффективный доступ к расширенной аналитике.
- Унифицированное хранилище данных для BI.
- Доступность исторических данных и версионирование.

Рассмотрим варианты использования:

1. Платформы обработки больших данных для ETL / ELT. Они требуют надежного и простого в обслуживании процесса ETL / ELT для подачи в них данных. Когда организации размещают расчеты в облаке они предпочитают использовать Cloud Object Storages для озера данных чтобы минимизировать затраты на хранение, а затем вычислять производные наборы данных в более оптимизированных системах хранилищ данных. Использование Delta Lake вместо отдельного озера и хранилища данных снижает задержку при запросе новых данных, устраняя необходимость в отдельном процессе трансформации между ними. Наконец, поддержка Delta Lake QL, программных

1. API через Apache Spark вводит унификацию в описании пайплайнов обработки данных.
2. Хранилища данных и BI. Традиционные системы хранилищ данных сочетают в себе функциональность ETL / ELT с эффективными инструментами QL из таблиц, чтобы рассчитывать BI. Столбчатые хранилища, кластеризация и индексирование, оптимизаторы – все важно для удобной работы с BI. Все это уже есть в дельта-таблицах. Одним из преимуществ запуска BI через Delta Lake является то, что аналитикам легче предоставить для работы свежие данные, поскольку данные не нужно загружать в отдельную систему.
3. Эффективный доступ к расширенной аналитике. DataFrames – основной тип данных, используемый для входных данных в библиотеках аналитики для Apache Spark, включая ML / MLlib, GraphFrames, PySpark и пр. Все расчеты могут использовать ускоренный ввод-вывод при наличии оптимизации в Delta Lake.
4. Планировщик запросов Spark помещает выборки и предикты непосредственно в класс модуля Data Source для каждого чтения источника данных. Например, если в Data Source для Delta Lake есть оптимизации кэширования, пропуска части данных для ускорения расчетов, то все это будет доступно во всех библиотеках.



Рассмотрим формат хранения дельта-таблицы. Каждая таблица хранится в каталоге файловой системы (здесь – “mytable”) или в виде объектов, начинающихся с одного и того же префикса ключа «каталога», в хранилище объектов. Содержимое таблицы “mytable” хранится в объектах Apache Parquet, организованных в каталоги с использованием соглашения об именовании разделов Hive. Таблица разделена по столбцу даты, поэтому объекты данных для каждой даты находятся в отдельных каталогах.

Parquet в качестве базового формата данных, поскольку он ориентирован на столбцы, предлагает разнообразные обновления сжатия, поддерживает вложенные типы данных для полуструктурированных

данных и уже имеет высокопроизводительные имплементации во многих механизмах. Использование существующего открытого формата файлов также позволило Delta Lake продолжать использовать преимущества недавно выпущенных обновлений библиотек Parquet и упростить разработку коннекторов для других движков. Другие форматы с открытым исходным кодом, такие как ORC, вероятно, работали бы аналогичным образом, но у Parquet была наиболее полная поддержка в Spark.

Журнал хранится в подкаталоге “_delta_log”. Он содержит последовательность объектов JSON с возрастанием, числовые идентификаторы с нулевым заполнением для хранения записей журнала, а также случайные контрольные точки для конкретных объектов журнала, которые суммируют журнал до этой точки в формате Parquet. Некоторые простые протоколы доступа (в зависимости от атомарных операций, доступных в каждом хранилище объектов) используются для создания новых записей журнала или контрольных точек и позволяют клиентам согласовывать порядок транзакций. Каждый объект записи журнала (например, 000003.json) содержит массив действий, которые следует применить к предыдущей версии таблицы для создания следующей.

Протоколы доступа Delta Lake предназначены для того, чтобы клиенты могли выполнять сериализуемые транзакции, используя только операции с Cloud Data Storage без опирания на гарантии провайдера. Например, объект записи журнала, такой как 000003.json, является корневой структурой данных, которая необходима клиенту для чтения определенной версии таблицы. Учитывая содержимое этого объекта, клиент может затем запросить другие объекты из Cloud Object Storage, заблокируясь, если они еще не видны из-за задержек консистентности. Очевидно, нужно гарантировать эксклюзивный лок на создание следующей записи журнала (например, 000003.json), а затем использовать это для имплементации оптимистического лока.

Пример создания таблицы Delta Lake из датафрейма

```
#1 Load a file into a dataframe and save it as a delta table
df = spark.read.load('/datalake/mydata.csv',format='csv',header=True)
dtable_path = "/deltalake/mydata"
df.write.format("delta").save(dtable_path)
#2 Replace or add rows to an existing DL table
new_df.write.format("delta").mode("overwrite").save(dtable_path)
new_rows_df.write.format("delta").mode("append").save(dtable_path)
```

Один из самых простых способов создать таблицу Delta Lake — сохранить датафрейм в дельта-таблице, указав путь, по которому должны храниться файлы данных и связанная с ними информация метаданных для таблицы.

Пример #1. Загрузить данные из существующего файла, а затем сохраняет этот датафрейм в новой папке в формате дельта-таблицы.

Пример #2. Заменить существующую дельта-таблицу содержимым датафрейма, используя режим перезаписи, как показано здесь. А также добавить строки из датафрейма в существующую дельта-таблицу, используя режим добавления.

Апдейты таблицы по условию и версионирование

```
#3 Create a deltaTable object and update the table with the Price -10%
deltaTable = DeltaTable.forPath(spark, dtable_path)
deltaTable.update(
    condition="group == 1",  set={ "price": "price * 0.9" })
#4 Retrieve data from a specific version
df = spark.read.format("delta")
    .option("versionAsOf", 0).load(dtable_path)
```

Рекомендуется делать вставку, обновление, удаление строк в существующей таблице как отдельные транзакционные операции. Чтобы внести такие изменения в таблицу Delta Lake можно использовать DeltaTable API.

Пример #3. Обновить столбец цен для всех строк по группе. Здесь все просто: обновляем по условию совпадения по “group”.

Пример #4. Извлекать предыдущие версии таблицы для историчности. Можно получить данные из определенной версии дельта-таблицы, указав требуемую версию на датафрейме в качестве опции “versionAsOf”.

Создание таблиц в Hive метасторе для SparkQL

```
#5 Save as an external table with path
df.write.format("delta")
    .option("path", dtable_path).saveAsTable("extTable")
spark.QL("CREATE TABLE extTable USING DELTA LOCATION '/mydata'")
#6 Create a catalog table using the DeltaTableBuilder
DeltaTable.create(spark).tableName("default.products") \
    .addColumn("id", "INT").addColumn("name", "STRING") \
    .addColumn("group", "STRING").addColumn("price", "FLOAT") \
    .execute()
```

Во всех примерах таблица создается без явной схемы. В случае таблиц, созданных путем записи фрейма данных, схема таблицы наследуется от фрейма данных. При создании внешней таблицы схема наследуется от любых файлов, которые в данный момент хранятся в расположении таблицы.

Пример #6. Однако при создании новой управляемой таблицы или внешней таблицы с пустым в данный момент местоположением вы определяете схему таблицы, указывая имена столбцов, типы и возможность определения значений “NULL” в составе инструкции “CREATE TABLE”.