Визуализация данных в реальном времени. Лямбда-архитектура

На этом уроке

1. Об отличиях пакетной и потоковой обработки данных.
2. Почему пакетная обработка данных подходит не для всех задач.
3. Технические особенности решения BI задачи на основе потока данных.

Теория

1. Отличия пакетного и потокового подходов к преобразованию данных.
2. Примеры задач на построение визуализации по данным в режиме реального времени.
3. Решения для построения визуализации по потоку данных.
4. Lambda- и Kappa-архитектуры.

Потоковая обработка данных

Традиционный ETL-процесс включает в себя фазу Extract (буквально по определению), которая посвящена извлечению данных из источника. Таким образом, **аналитическое хранилище всегда отстает от источника** при использовании такого подхода. Отставание может составлять часы, сутки и даже недели и месяцы в зависимости от частоты выполнения ETL-процесса.

Такая модель распространения данных называется *pull*-моделью (от англ. pull — тянуть): данные «затягиваются» в хранилище ETL-процессом. Однако **источник может поставлять данные в хранилище самостоятельно**, реализуя *push*-модель распространения данных (от англ. push — толкать). Например, при каждом обновлении данных в «боевой» БД отправлять информацию об обновлении в аналитическую инфраструктуру.

При таком подходе **источник создает поток данных**. В традиционном подходе пакетной (англ. batch) обработки поток обрабатываемых данных ограничен — например, таблицей или партицией таблицы, которые выступают, по сути, «пакетом» данных. В случае же потоковой обработки **поток данных является неограниченным источником данных** (англ. unbounded source), а таблица является ограниченным источником (англ. bounded) данных.

Технически принципы работы с такими источниками концептуально отличаются: в случае пакетной обработки один шаг преобразований обрабатывает конечный объем данных, тогда как **обработка потока данных происходит непрерывно**.

Однако логически обработка потока данных и ограниченной таблицы могут не отличаться значительно: оба формата представляют набор событий, т.е. фактов, привязанных ко времени. Поэтому **те же преобразования, которые выполняются над данными в случае пакетной обработки, возможны и в потоковом случае**: например, событие-транзакция может содержать идентификатор пользователя, по которому можно сопоставить транзакции измерение-регион или измерение-возраст пользователя.

Отличия будут присутствовать на уровне инфраструктуры данных. Если описанную выше операцию («притягивание» информации о пользователе к транзакции) выполнять над пакетом данных, оптимальным будет использование реляционных СУБД. Записи в потоке данных же могут обогащаться индивидуально, поэтому для такого случае лучше подойдет in-memory key-value хранилище (впрочем, на уровне реализации реляционная СУБД может использовать схожий подход для сопоставления записей).

Как и в случае традиционного ETL, который преобразует данные для использования в аналитических задачах, потоковая обработка данных является методом для обеспечения потоковой аналитики. Данные из

обработанного потока визуализируются с использованием средств BI и используются в системах поддержки принятия решений.

Примеры задач на потоковую аналитику

Потоковая аналитика полезна в отраслях, где требуется оперативное принятие решений, чаще всего с участием человека. Задача похожа на технический мониторинг (например, нагрузки на сайт), однако ориентирована на бизнес-метрики.

Примером домена, в котором применима потоковая аналитика, является логистика: менеджер магазина или склада может следить за заполненностью полок, количеством активных сотрудников и другими показателями, на основе которых можно принимать решения «в моменте».

Маркетинг в офлайн-ритейле также может быть основан на потоковой аналитике. Например, агентства недвижимости могут подбирать клиенту персонализированные предложения в момент визита, чтобы побудить покупателя к действию.

Информацию о клиента в таком случае можно почерпнуть по данным, собранным устройствами-«маяками», которые «узнают» устройство пользователя, когда оно попадает в радиус действия их сети. Например, смартфон пользователя может выполнить запрос доступных сетей WiFi, передав свой MAC-адрес, по которому связанная рекламная площадка предоставит магазину связанные с клиентом атрибуты: покупательную способность, класс дохода, место проживания — эти факторы далее используются при выборе подходящего предложения.

Потоковая аналитика также обобщается на случай автоматизированного принятия решений на основе потока данных.

Другая сфера применения потоковой аналитики — информационная безопасность. Отслеживание подозрительной активности пользователей в режиме реального времени позволяет блокировать доступ злоумышленникам для избежания, например, DDoS-атак.

Аналогично примеру со сферой логистики поток данных может быть использован на производстве: в эпоху активного распространения интернета вещей (англ. Internet of Things, IoT) производственные помещения и станки активно снабжаются различными сенсорами, которые отслеживают, например, температуру, давление и прочие характеристики для предупреждения аварий и поломок. Решение о предотвращении критической ситуации может быть принято как человеком, так и автоматизированно.

Другой пример — сбор информации о пользователе в момент посещения им веб-сайта, например, интернет-магазина. Анализ поведения на основе данных о просмотрах товаров, добавлении в корзину, удалении из нее и исторической сводке о поведении пользователя позволяют реактивировать пользователей, конвертируя их просмотры в повторный заказ.

Реальный пример использования потоковой аналитики: 20% населения Нидерландов живут на территориях ниже уровня моря, и еще 50% — не выше одного метра над уровнем моря. Правительство в реальном времени отслеживает состояние транспортной сети, в буквальном смысле управляя даже светофорами, чтобы на основе сенсоров с дамб разгружать участки дорог, которые в текущий момент наиболее уязвимы в случае прорыва плотины.

Преимущества потоковой обработки данных

Выбор потокового подхода к работе с данными в примерах выше обусловлен преимуществами потоковой обработки над традиционной пакетной обработкой:

1. Данные доступны для анализа со значительно меньшим отставанием от источника данных.
2. Аналитик работает с актуальными данными, которые постоянно в актуальном состоянии, и может принимать решения в день происхождения события.

3. Обогащение данных в реальном времени позволяет воссоздать «контекст» клиента и уточнять рекомендации «в моменте».

Недостатки и технические сложности реализации потоковой обработки данных

В этом разделе потоковая обработка данных сравнивается с традиционной пакетной.

Основной недостаток потокового подхода к обработке заключается в том, что **не все источники поддерживают push-модель**. В случае реализации традиционного ETL-процесса Data Engineer самостоятельно контролирует процесс поставки данных в хранилище: задействовать ресурс разработчиков со стороны источника данных не требуется.

Обычно ETL-процессы извлекают данные из «боевых» баз данных. Данные из таких БД интуитивно забирать порциями («пакетами»), и часто нет возможности получать из них обновления в потоковом режиме, потому как они не предусмотрены для этого. **Функциональность, позволяющая представить обновления состояния OLTP базы данных как поток событий, называется change data capture, CDC**. Эта опция все чаще появляется в современных базах данных, однако отключена по умолчанию, т.к. требует дополнительного обслуживания со стороны СУБД.

Другая сложность реализации потоковой обработки данных заключается в более жестких требованиях доступности: **инфраструктура данных должна принимать данные непрерывно**, без сбоев. Потоковая аналитика остро реагирует на задержки, т.к. непрерывность поставки и актуальность данных является одним из ключевых ее преимуществ.

Задача упрощается, если инфраструктура данных самого приложения (не аналитическая) реализует event-driven парадигму. В ядре такого приложения обычно находится шина или брокер сообщений, читать события из которого можно с произвольного требуемого момента времени (обычно — с последнего прочитанного сообщения). Таким образом, такой подход объединяет преимущества pull и push моделей распространения данных: если аналитическая инфраструктура будет недоступна, добрать события получится при следующем запуске.

Визуализация потоковых данных также формирует ряд требований к промежуточному хранилищу данных, на основе которого работает BI инструмент. Класс подходящих для этих целей решений называется fast data sinks. Они **поддерживают быструю запись и множество параллельных подключений на чтение**. Хорошими примерами таких решений выступают файлы в файловой системе Hadoop (HDFS) в специальных эффективных форматах Kudu и Parquet, программное обеспечение, ориентированное на поиск (например, Elasticsearch, Cloudera Search и Solr), in-memory БД как MemSQL и хранилища данных как Snowflake.

Fast data sinks — это не ограниченный класс решений, а набор требований к ним. Инструменты потоковой аналитики непрерывно выполняют запросы (англ. continuous queries) к такому источнику, что и формирует перечисленные требования.

Данные попадают в fast data sink через коннекторы из источников данных, формирующих поток: Kafka Connect, Google Cloud Dataflow, Apache NiFi, Spark Streaming, SQLStream. Данные, поступающие в fast data sink могут быть уже обогащены, как было описано выше (см. пример с «притягиванием» пользователя к транзакции).

BI-решения, подключившись к fast data sink, регулярно опрашивают его на предмет обновленных данных, что и создает высокую нагрузку на чтение.

Альтернативным вариантом было бы подключение напрямую к потоку данных из BI инструмента, однако в более широком класс BI задач, чем визуализация потока данных, может требоваться доступ в том числе к историческим данным. Тогда как источник потока данных хранит лишь некоторую ограниченную историю событий в детальном виде, fast data sink может обслуживать в том числе запросы на аналитику на основе полной истории.

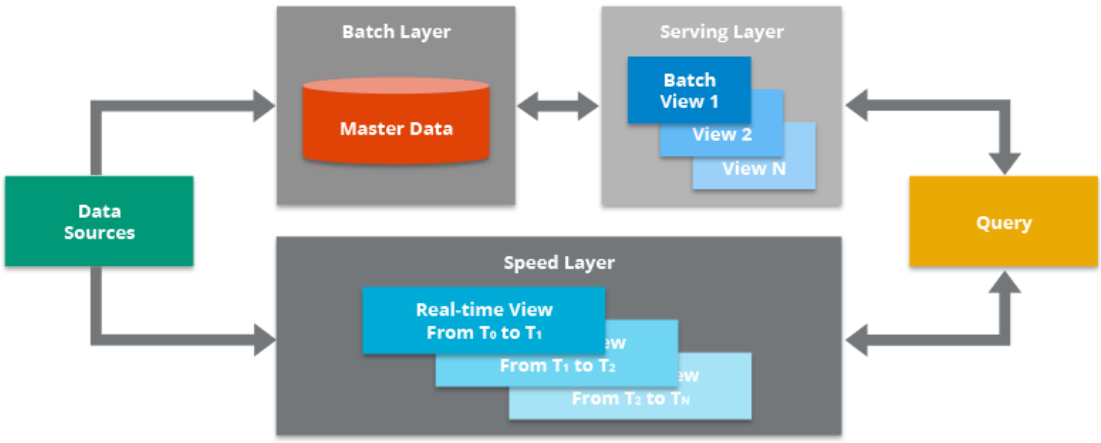
Гибридные архитектуры

Ограничением потоковой обработки выступает сложность вычислений, которые можно производить над потоком данных. Обычно она значительно уступает возможностям, которые обеспечивает пакетная обработка. Преобразования чаще носят «потоковый характер»: к потоку данных может быть применена фильтрация, простые построчные преобразования (каждое событие из потока трансформируется индивидуально).

Агрегация также доступна и обычно производится над временнЫм окном: например, агрегируются данные за последнюю неделю или час, или же агрегат может обновлять состояние с начала потока данных (например, счетчики). Однако вычислить агрегат за всю историю в потоковом режиме может быть невозможно из-за объема данных.

В качестве решения используется **гибридный подход, который совмещает достоинства потоковой и пакетной обработки данных**. Известная архитектура, реализующие такой подход, называется Lambda.

Lambda-архитектура расщепляет поток событий на две копии: первая копия наполняет решение для потоковой аналитики данных (и хранит ограниченную историю), а вторая — традиционное хранилище данных (хранящее полную историю). На мощностях хранилища регулярно (например, раз в час или день) выполняются ресурсоемкие вычисления: подсчет уникальных пользователей, множественные JOIN'ы. Результирующие агрегаты размещаются в fast data sinks и надстраиваются из потока данных в режиме реального времени. Таким образом, **более свежие данные обогащают более точные**:



Опираясь на родственность потока данных и таблицы фактов, некоторые решения успешно совмещают подходы в унифицированной модели программирования, давая разработчикам возможность работать как с потоковыми данными, так и с пакетными, переиспользуя логику обработки. Код для обработки данных в буквальном смысле остается тем же самым, меняется только источник (и его природа).

Одним из таких решений является бессерверный Google Cloud Dataflow (или open source версия его SDK — Apache Beam). Но важно помнить, что потоковая обработка имеет ограничения на сложность вычислений, описанные в разделе этого урока о технических сложностях реализации. Dataflow также поддерживает промежуточный подход к потоковой обработке — поток данных разбивается на окна ограниченного размера, что позволяет выполнять над ними сложный вычисления уровня, доступного для пакетной обработки. Dataflow самостоятельно определяет «ограниченность» источника данных и унифицирует подход к обработке.

Пример чтения данных с использованием Dataflow:

```
def main(arvg=None):
      parser = argparse.ArgumentParser()
      parser.add_argument(
          '--input-file',
          dest='input_file',
          default='/Users/home/words-example.txt',
      )
      known_args, pipeline_args = parser.parse_known_args(argv)
      pipeline_options = PipelineOptions(pipeline_args)
      p = beam.Pipeline(options=pipeline_options)
      lines  = p | 'read' >> ReadFromText(known_args.input_file)
```

Пример выше ориентирован на пакетную обработку, т.к. по умолчанию читает данные из файла. Примерно источника, который создает неограниченный поток данных, выступает Google Cloud PubSub — универсальный сервис очередей сообщений. Чтение из него не отличается значительно от чтения из файла:

```
data = p | beam.io.ReadFromPubSub(topic=known_args.input_topic)
lines = data | 'DecodeString' >> beam.Map(lambda d: d.decode('utf-8'))
```

Следующим образом реализуется разбиение потока на окна:

```
windowed_words = input | beam.WindowInto(window.FixedWindows(60 *
window_size_minutes))
```

Непосредственно обработка данных в Apache Beam на примере задачи подсчета слов реализуется следующим образом:

```
class CountWords(beam.PTransform):
  def expand(self, pcoll):
    return (
        pcoll
        # Convert lines of text into individual words.
        | 'ExtractWords' >>
        beam.FlatMap(lambda x: re.findall(r'[A-Za-z\']+', x))

        # Count the number of times each word occurs.
        | beam.combiners.Count.PerElement())

counts = lines | CountWords()
```

Источники

1. Streaming Analytics Use Cases
2. Streaming Data Analytics for Fast BI Insights
3. Streaming Analytics: The Value is in the Action
4. What Is Lambda Architecture?
5. Google Cloud Dataflow
6. Apache Beam word count example