5.3.1. Агрегирование объема продаж акций по отраслям промышленности

Агрегирование и группировка — жизненно необходимые инструменты при работе с потоковыми данными. Исследования отдельных записей по мере поступления часто оказывается недостаточно. Для извлечения из данных дополнительной информации необходимы их группировка и комбинирование.

В этом примере вам предстоит примерить костюм внутридневного трейдера, которому нужно отслеживать объемы продаж акций компаний в нескольких отраслях промышленности. В частности, вас интересуют пять компаний с наибольшими объемами продаж акций в каждой из отраслей промышленности.

Для подобного агрегирования потребуется несколько следующих шагов по переводу данных в нужный вид (если говорить в общих чертах).

1. *Создать источник на основе топика, публикующий необработанную информацию по торговле акциями*. Нам придется отобразить объект типа `StockTransaction` в объект типа `ShareVolume`. Дело в том, что объект `StockTransaction` содержит метаданные продаж, а нам нужны только данные о количестве продаваемых акций.
2. *Сгруппировать данные* `ShareVolume` *по символам акций*. После группировки по символам можно свернуть эти данные до промежуточных сумм объемов продаж акций. Стоит отметить, что метод `KStream.groupBy` возвращает экземпляр типа `KGroupedStream`. А получить экземпляр `KTable` можно, вызвав далее метод `KGroupedStream.reduce`.

Что такое интерфейс KGroupedStream

Методы `KStream.groupBy` и `KStream.groupByKey` возвращают экземпляр `KGroupedStream`. `KGroupedStream` является промежуточным представлением потока событий после группировки по ключам. Он вовсе не предназначен для непосредственной работы с ним. Вместо этого `KGroupedStream` используется для операций агрегирования, результатом которых всегда является `KTable`. А поскольку результатом операций агрегирования является `KTable` и в них применяется хранилище состояния, то, возможно, не все обновления в результате отправляются дальше по конвейеру.

Метод `KTable.groupBy` возвращает аналогичный `KGroupedTable` — промежуточное представление потока обновлений, перегруппированных по ключу.

Сделаем небольшой перерыв и посмотрим на рис. 5.9, на котором показано, чего мы добились. Эта топология должна быть вам уже хорошо знакома.

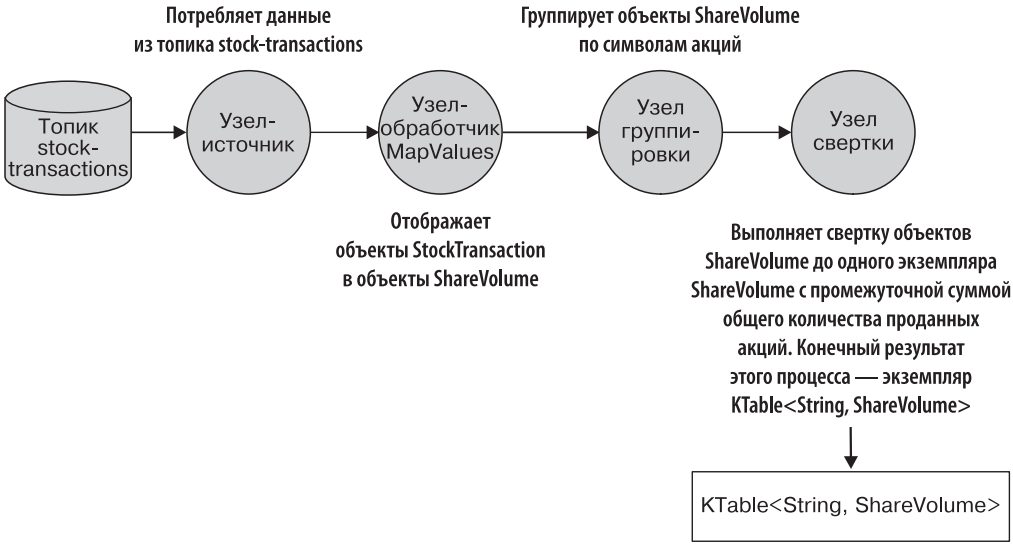

Рис. 5.9. Отображение и свертка объектов StockTransaction в объекты ShareVolume с последующей сверткой их до промежуточных сумм

Взглянем теперь на код для этой топологии (его можно найти в файле src/main/java/bbejeck/chapter_5/AggregationsAndReducingExample.java) (листинг 5.2).

Листинг 5.2. Источник для преобразования-свертки биржевых транзакций

```
KTable<String, ShareVolume> shareVolume =
➥ builder.stream(STOCK_TRANSACTIONS_TOPIC,
                  Consumed.with(stringSerde, stockTransactionSerde)
➥ .withOffsetResetPolicy(EARLIEST))
➥ .mapValues(st -> ShareVolume.newBuilder(st).build())
➥ .groupBy((k, v) -> v.getSymbol(),
                    Serialized.with(stringSerde, shareVolumeSerde))
➥ .reduce(ShareVolume::reduce);
```

Узел-источник потребляет данные из топика

Отображает объекты StockTransaction в объекты ShareVolume

Выполняет свертку объектов ShareVolume до промежуточного количества проданных акций

Группирует объекты ShareVolume по символам акций

Приведенный код отличается краткостью и большим объемом производимых в нескольких строках действий. В первом параметре метода `builder.stream` вы можете заметить нечто новое для себя: значение перечисляемого типа `AutoOffsetReset.EARLIEST` (существует также и `LATEST`), задаваемое с помощью метода `Consumed.withOffsetResetPolicy`. С помощью этого перечисляемого типа можно указать стратегию сброса смещений для каждого из `KStream` или `KTable`, он обладает приоритетом над параметром сброса смещений из конфигурации.

GroupByKey и GroupBy

В интерфейсе `KStream` есть два метода для группировки записей: `GroupByKey` и `GroupBy`. Оба возвращают `KGroupedTable`, так что у вас может появиться закономерный вопрос: в чем же различие между ними и когда использовать какой из них?

Метод `GroupByKey` применяется, когда ключи в `KStream` уже непустые. А главное, флаг «требует повторного секционирования» никогда не устанавливался.

Метод `GroupBy` предполагает, что вы меняли ключи для группировки, так что флаг повторного секционирования установлен в `true`. Выполнение после метода `GroupBy` соединений, агрегирования и т. п. приведет к автоматическому повторному секционированию.

Резюме: следует при малейшей возможности использовать `GroupByKey`, а не `GroupBy`.

Что делают методы `mapValues` и `groupBy` — понятно, так что взглянем на метод `sum()` (его можно найти в файле src/main/java/bbejeck/model/ShareVolume.java) (листинг 5.3).

Листинг 5.3. Метод ShareVolume.sum

```
public static ShareVolume sum(ShareVolume csv1, ShareVolume csv2) {
      Builder builder = new Builder(csv1);
      builder.shares = csv1.shares + csv2.shares;
      return builder.build();
}
```

Используем конструктор копирования класса Builder

Устанавливаем количество акций равным сумме количеств акций из обоих объектов ShareVolume

Вызываем метод build() и возвращаем новый объект ShareVolume

ПРИМЕЧАНИЕ

Мы уже встречали паттерн проектирования «Строитель» ранее в этой книге, но здесь он используется несколько в другом контексте. В данном примере «Строитель» применяется для создания копии объекта и обновления поля без модификации исходного объекта.

Метод `ShareVolume.sum` возвращает промежуточную сумму объема продаж акций, а результат всей цепочки вычислений представляет собой объект `KTable<String, ShareVolume>`. Теперь вы понимаете, какую роль играет `KTable`. При поступлении объектов `ShareVolume` в соответствующем объекте `KTable` сохраняется последнее актуальное обновление. Важно не забывать, что все обновления отражаются в предшествующем `shareVolumeKTable`, но не все отправляются далее.

ПРИМЕЧАНИЕ

Зачем выполнять свертку вместо агрегирования? Хотя свертка — одна из разновидностей агрегирования, при ней вы получаете объект того же типа. Хотя операция агрегирования также суммирует значения, но может возвращать объект другого типа.

Далее с помощью этого `KTable` мы выполняем агрегирование (по количеству продаваемых акций), чтобы получить пять компаний с наибольшими объемами продаж акций в каждой из отраслей промышленности. Наши действия при этом будут аналогичны действиям при первом агрегировании.

1. Выполнить еще одну операцию `groupBy` для группировки отдельных объектов `ShareVolume` по отраслям промышленности.
2. Приступить к суммированию объектов `ShareVolume`. На этот раз объект агрегирования представляет собой очередь по приоритету фиксированного размера. В такой очереди фиксированного размера сохраняются только пять компаний с наибольшими количествами проданных акций.
3. Отобразить очереди из предыдущего пункта в строковое значение и вернуть пять наиболее продаваемых по количеству акций по отраслям промышленности.
4. Записать результаты в строковом виде в топик.

На рис. 5.10 показан граф топологии движения данных. Как вы видите, второй круг обработки достаточно прост.

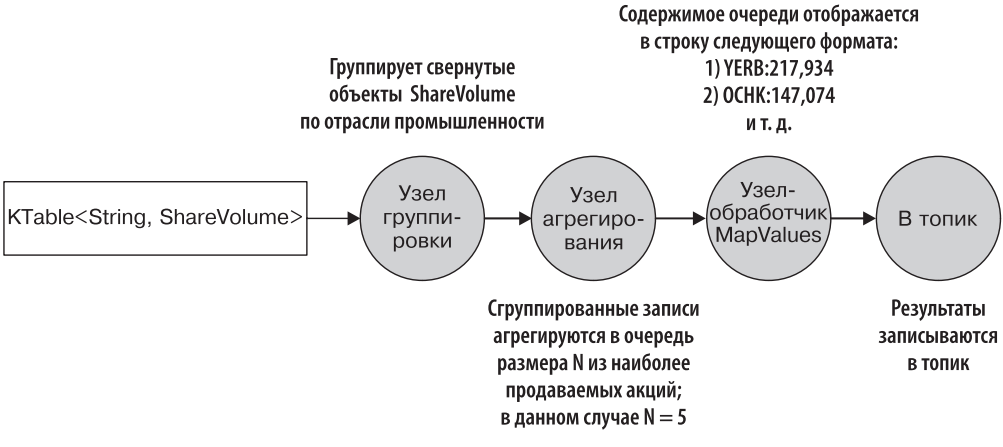

Рис. 5.10. Топология для группировки по отрасли промышленности, агрегирования для получения пяти самых продаваемых, отображения очереди из этих пяти в строковое значение и записи данной строки в топик

Теперь, четко уяснив себе структуру этого второго круга обработки, можно обратиться к его исходному коду (вы найдете его в файле src/main/java/bbejeck/chapter_5/AggregationsAndReducingExample.java) (листинг 5.4).

В данном инициализаторе есть переменная `fixedQueue`. Это пользовательский объект — адаптер для `java.util.TreeSet`, который применяется для отслеживания *N* наибольших результатов в порядке убывания количества проданных акций.

Листинг 5.4. KTable: группировка и агрегирование

```
Comparator<ShareVolume> comparator =
➥ (sv1, sv2) -> sv2.getShares() - sv1.getShares()

FixedSizePriorityQueue<ShareVolume> fixedQueue =
➥ new FixedSizePriorityQueue<>(comparator, 5);

shareVolume.groupBy((k, v) -> KeyValue.pair(v.getIndustry(), v),
➥ Serialized.with(stringSerde, shareVolumeSerde))
  .aggregate(() -> fixedQueue,
             (k, v, agg) -> agg.add(v),
             (k, v, agg) -> agg.remove(v),
             Materialized.with(stringSerde, fixedSizePriorityQueueSerde))
  .mapValues(valueMapper)
  .toStream().peek((k, v) ->
➥ LOG.info("Stock volume by industry {} {}", k, v))
  .to("stock-volume-by-company", Produced.with(stringSerde,
➥ stringSerde));
```

aggregate инициализируется экземпляром класса FixedSizePriorityQueue (исключительно в демонстрационных целях!)

Группирует по отрасли промышленности, передавая нужные объекты Serde

Добавляем новые обновления с помощью метода добавления

Удаляем старые обновления с помощью метода удаления

Объект Serde для агрегирования

Экземпляр valueMapper преобразует результат агрегирования в возвращаемое далее строковое значение

Вызываем метод toStream() для журналирования результатов (в консоль) с помощью метода peek

Записываем результаты в топик stock-volume-by-company

Вы уже встречались с вызовами `groupBy` и `mapValues`, так что не будем на них останавливаться (мы вызываем метод `KTable.toStream`, поскольку метод `KTable.print` считается устаревшим). Но вы пока еще не видели `KTable`-версию метода `aggregate()`, так что мы потратим немного времени на его обсуждение.

Как вы помните, `KTable` отличает то, что записи с одинаковыми ключами считаются обновлениями. `KTable` заменяет старую запись новой. Агрегирование происходит подобным же образом: агрегируются последние записи с одним ключом. При поступлении записи она добавляется в экземпляр класса `FixedSizePriorityQueue` с помощью сумматора (второй параметр в вызове метода `aggregate`), но если уже существует другая запись с тем же ключом, то старая запись удаляется с помощью вычитателя (третий параметр в вызове метода `aggregate`).

Это все значит, что наш агрегатор, `FixedSizePriorityQueue`, вовсе не агрегирует *все* значения с одним ключом, а хранит скользящую сумму количеств N наиболее продаваемых видов акций. В каждой поступающей записи содержится общее количество проданных до сих пор акций. `KTable` даст вам информацию о том, акций каких компаний продается больше всего в настоящий момент, скользящее агрегирование каждого из обновлений не требуется.

Мы научились делать две важные вещи:

- группировать значения в `KTable` по общему для них ключу;
- выполнять над этими сгруппированными значениями такие полезные операции, как свертка и агрегирование.

Умение выполнять эти операции важно для понимания смысла данных, движущихся через приложение Kafka Streams, и выяснения того, какую информацию они несут.

Мы также соединили воедино некоторые из ключевых понятий, обсуждавшихся ранее в этой книге. В главе 4 мы рассказывали, насколько важно для потокового приложения отказоустойчивое, локальное состояние. Первый пример из этой главы продемонстрировал, почему настолько важно локальное состояние — оно дает возможность отслеживать, какую информацию вы уже видели. *Локальный* доступ позволяет избежать сетевых задержек, благодаря чему приложение становится более производительным и устойчивым к ошибкам.

При выполнении любой операции свертки или агрегирования необходимо указать название хранилища состояния. Операции свертки и агрегирования возвращают экземпляр `KTable`, а `KTable` использует хранилище состояния для замены старых результатов новыми. Как вы видели, далеко не все обновления отправляются далее по конвейеру, и это важно, поскольку операции агрегирования предназначены для получения итоговой информации. Если не применять локальное состояние, `KTable` будет отправлять дальше *все* результаты агрегирования и свертки.

Далее мы посмотрим на выполнение таких операций, как агрегирование, в пределах конкретного промежутка времени — так называемых *оконных операций* (windowing operations).

5.3.2. Оконные операции

В предыдущем разделе мы познакомились со «скользящими» сверткой и агрегированием. Приложение производило непрерывную свертку объема продаж акций с последующим агрегированием пяти наиболее продаваемых на бирже акций.

Иногда подобные непрерывные агрегирование и свертка результатов необходимы. А иногда нужно выполнить операции только над заданным промежутком времени. Например, вычислить, сколько было произведено биржевых операций с акциями конкретной компании за последние 10 минут. Или сколько пользователей нажало на новый рекламный баннер за последние 15 минут. Приложение может производить такие операции многократно, но с результатами, относящимися только к заданным промежуткам времени (временным окнам).

Подсчет биржевых транзакций по покупателю

В следующем примере мы займемся отслеживанием биржевых транзакций по нескольким трейдерам — либо крупным организациям, либо смышленым финансистам-одиночкам.

Существует две возможные причины для подобного отслеживания. Одна из них — необходимость знать, что покупают/продают лидеры рынка. Если эти крупные игроки и искушенные инвесторы видят для себя открывающиеся возможности,

имеет смысл следовать их стратегии. Вторая причина заключается в желании заметить любые возможные признаки незаконных сделок с использованием внутренней информации. Для этого вам понадобится проанализировать корреляцию крупных всплесков продаж с важными пресс-релизами.

Такое отслеживание состоит из таких этапов, как:

- создание потока для чтения из топика `stock-transactions`;
- группировка входящих записей по идентификатору покупателя и биржевому символу акции. Вызов метода `groupBy` возвращает экземпляр класса `KGroupedStream`;
- возвращение методом `KGroupedStream.windowedBy` потока данных, ограниченного временным окном, что позволяет выполнять оконное агрегирование. В зависимости от типа окна возвращается либо `TimeWindowedKStream`, либо `SessionWindowedKStream`;
- подсчет транзакций для операции агрегирования. Оконный поток данных определяет, учитывается ли при этом подсчете конкретная запись;
- запись результатов в топик или вывод их в консоль во время разработки.

Топология данного приложения проста, но наглядная ее картинка не помешает. Взглянем на рис. 5.11.

Далее мы рассмотрим функциональность оконных операций и соответствующий код.

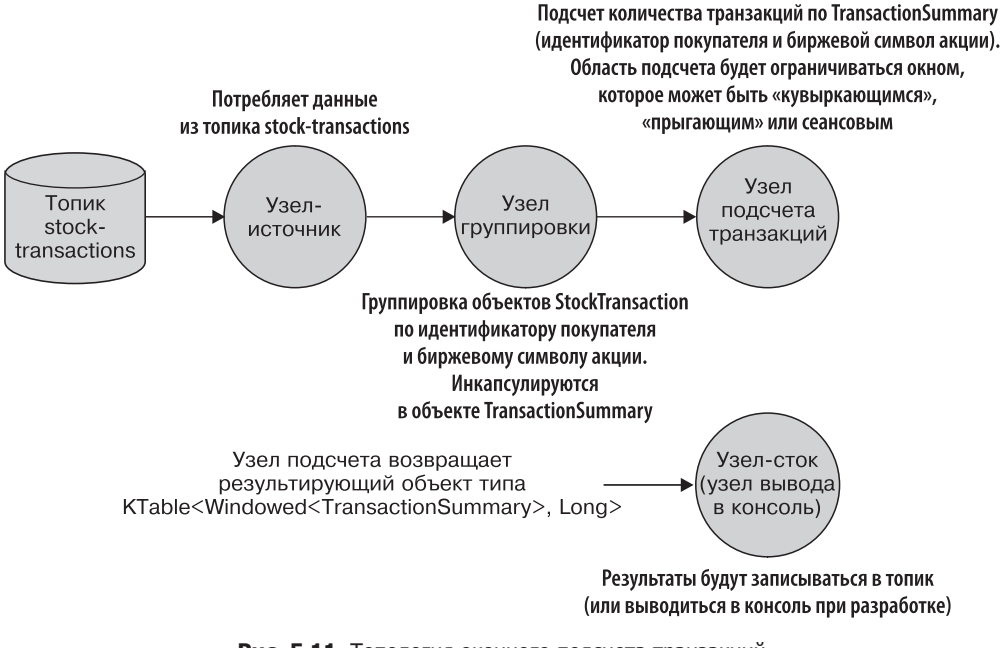

Рис. 5.11. Топология оконного подсчета транзакций

Типы окон

В Kafka Streams существует три типа окон:

- сеансовые;
- «кувыркающиеся» (tumbling);
- скользящие/«прыгающие» (sliding/hopping).

Какое выбрать — зависит от бизнес-требований. «Кувыркающиеся» и «прыгающие» окна ограничиваются по времени, в то время как ограничения сеансовых связаны с действиями пользователей — длительность сеанса (-ов) определяется исключительно тем, насколько активно ведет себя пользователь. Главное — не забывать, что все типы окон основываются на метках даты/времени записей, а не на системном времени.

Далее мы реализуем нашу топологию с каждым из типов окон. Полный код будет приведен только в первом примере, для других типов окон ничего не изменится, кроме типа оконной операции.

Сеансовые окна

Сеансовые окна сильно отличаются от всех остальных типов окон. Они ограничиваются не столько по времени, сколько активностью пользователя (или активностью той сущности, которую вы хотели бы отслеживать). Сеансовые окна разграничиваются периодами бездействия.

Рисунок 5.12 иллюстрирует понятие сеансовых окон. Меньший сеанс будет сливаться с сеансом слева от него. А сеанс справа будет отдельным, поскольку следует за длительным периодом бездействия. Сеансовые окна основываются на действиях пользователей, но применяют метки даты/времени из записей для определения того, к какому сеансу относится запись.

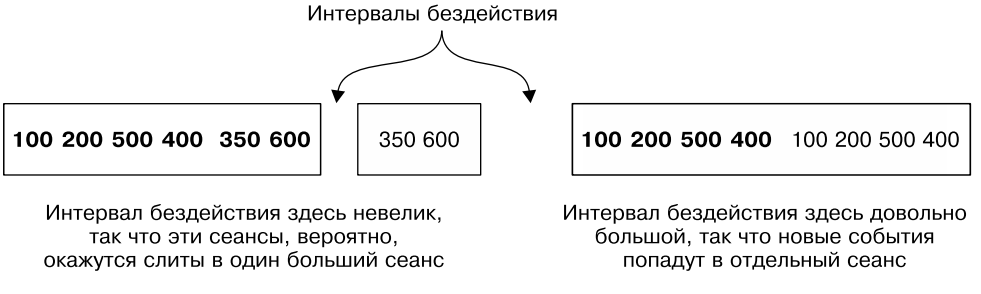

Рис. 5.12. Сеансовые окна, разделяемые коротким периодом бездействия, объединяются в новый, более крупный сеанс