FlinkSQL

The SQL dialect for Flink

Khuong Duy Vu

Department of Informatics

ELTE University

This thesis is submitted for the degree of

Master Thesis

May 2015

Chapter 1

Introduction(3 pages)

- BUILDING ALGEBRA SEMANTIC FOR FLINK STREAMING - IMPLEMENTATION

Big Data

Data Streaming

CQL and related work

Features of stream processing languages(Fundamental of Stream Processing book page 110)

Flink and FlinkQL

Structure

[2014] Fundamentals of Stream Processing- Application Design, Systems, and Analytics.pdf [2]

Sliding Window Query Processing over Data Stream by Lukasz Golab

Chapter 2

Data Stream Model(10 pages)

Order

Ordering is rather important in Distributed system, specially Stream Processing System in particular.

In traditional model, there are a single program, one process, one memory space running on one CPU. Programs are written to be executed in an ordered fashion in queue: staring from the beginning, and then going toward the last. In distributed system, programs run in many machines across the network but try to reserve the order of the result. One of easiest way to define "correctness" is to say "it works like it would on a single machine". And that usually means that a) we run the same operations and b) that we run them in the same order - even if there are multiple machines. [Takada]

In theory, they are defined 2 types of orders: total order and partial order.

Definition 2.1 *Paritial order is a binary relation $\leq$ over a set P which is reflexive, anti-symmetric and transitive, i.e., which satisfies for all a, b and c in S (wiki):*

- *$a \leq a$ (reflexivity)*
- *if $a \leq b$ and $b \leq a$ then $a = b$ (antisymmetry)*
- *if $a \leq b$ and $b \leq c$ then $a \leq c$ (transitivity)*

Definition 2.2 *Total order is a binary relation "$\leq$" over a set S which is anti-symmetric and transitive and total. Therefore, total order is a partial order with totality*

- *$a \leq b$ or $b \leq a$ (totality)*

Total order “<” is strict on a set S if and only if $(S,<)$ has no non-comparable pairs:

$$\forall x,y \in S \Rightarrow x<y \cup y<x \tag{2.1}$$

In a totally ordered set, every two elements are comparable whereas in a partial ordered set, some pairs of elements are incomparable and hence we do not have the exact order of every item.

The natural state in a distributed system is partial order. Neither the network nor independent nodes make any guarantees about relative order; but at each node, you can observe a local order.

In streaming processing, one may discretize a stream into a discrete stream of windows. Therefore, we need to specify the order of elements to define which elements should be inserted into a given window. Furthermore, data stream involve the temporal order and the positional order [11] in stream.

The temporal order is induced by the timestamp of items in a stream. Using the value of timestamp, one can determine whether something happen chronologically before something else. Nevertheless, if some items happen simultaneously, they will have the same timestamp, then it may not be strict. For instance, there are two items with the same timestamp but system is required to take one only, the chosen is non-deterministic between two. Therefore, we may require a strict order.

The positional is a strict order induced by the position of item in stream. Two items may have the same timestamp but one may arrive before the other so that they have different positional orders. The positional order can be defined by arrival order or id of item regardless of explicit timestamp.

//TODO:The temporal order: Order. When I say that time is a source of order, what I mean is that:

we can attach timestamps to unordered events to order them we can use timestamps to enforce a specific ordering of operations or the delivery of messages (for example, by delaying an operation if it arrives out of order) we can use the value of a timestamp to determine whether something happened chronologically before something else.

Time

Time Domain The time domain $\mathbb{T}$ is a discrete, total ordered, countably infinite set of time instants $t \in \mathbb{T}$. We assume that $\mathbb{T}$ is bounded in the past , but not necessarily in the future. For the sake of simplicity, we will assume that the time domain is the domain of non-negative long integer ($\mathbb{T} = \mathbb{N}$) 0,1,2,3,..[6] and totally ordered.

2 type of times: system time t^{sys} (implicit) and application time t^{app} (explicit). These both take values from the time domain $\mathbb{T}$, but carry two different meaning.

Application Timestamps In many case, each item in stream contains an explicit source-assigned timestamp itself. In other words, the timestamp attribute will be a part of the stream schema. Consider a common log format for web application which contains a timestamp specifying when the action is taken place. A log line record user named *pablo* to get an image

```
216.58.209.174 user-identifier pablo [10/Oct/2000:13:55:36 -0700] \\
"GET /image.gif HTTP/1.0" 200 1234
```

Sequence number attribute could represent a timestamp instead.

System Timestamps Even if the item arrives at the system are not equipped with a timestamp, the system assigns a timestamp to each tuple, by default using the system's local clock. While this process generates a raw stream with system timestamps that can be processed like a regular raw stream with application timestamps, the user should be aware that application time and system time are not necessarily synchronized[8]. Since system timestamp is assigned implicitly by system , one may not notice it presence on schema. As we mentions above, time domain is total ordered in local machine , but partial ordered across the system because of possible postpone on processing or asynchronous timestamp at different node.

Both application and system timestamp captures time information but they cary two different meanings. The former is related to the occurrence of the application event(when the event happens), whereas the latter is related to the occurrence of related system(when the corresponding event data arrive at system). Multiple items may have the same application timestamps but they will not arrive in the same order. Therefore, system will assign the different unique system timestamp based on their arrival. Sytem then can believe in the system timestamp as a strict total ordered basis for reasoning about arrival items to perform processing. For example, another log from different users arrive at system:

```
219.53.210.143 user-identifier fabio [10/Oct/2000:13:55:36 -0700] \\
"GET /image.gif HTTP/1.0" 200 1432
```

However, it might arrive after the first log for user *pablo* then system would response *pablo*'s first, instead of *fabio*'s request.

Note: more on Timestamp in Streams [4] page 13

Tuple

A tuple is a finite sequence of atomic values. Each tuple can be defined by a Schema corresponding to a composite type. Tuple can represend a relational tuple , a event or a record of sensor data and so on. [3]. For instance, the log format in previous example follow a schema:

```
<SourceIP, IdentityType, user, timestamp, action, response, packageSize >
```

A data tuple is the fundamental, or atomic data item, embedded in a data stream and processed by an application. A tuple is similar to a database row in that it has a set of named and typed attributes. Each instance of an attribute is associated with a value[2]. Furthermore, one can consider a tuple as a partial order mapping a finite subset of attribute names to atomic values[11]. A tuple consits of a set of (Atribute x Value) pairs such as $(SourceIP, 219.53.210.143)$

2.1 Stream Model

Based on time and tuple domain, basically, CQL [3] defines a data stream as

Definition 2.3 *A stream $\mathbb{S}$ is a countably continuous and infinite set of elements $< s,t > \in \mathbb{S}$, where s is a tuple belonging to the schema of $\mathbb{S}$ and $t \in \mathbb{T}$ is the timestamp of the element.*

There are several stream definitions varying based on the execution model of systems. On the previous definition, a timestamp attribute can be an non-strict total ordered application timestamp so that system may not rely on it to order the coming tuples to react on them. For this reason, stream can contain an extra physical identifier φ [10] such as increment tuple id to specify its order. The tuple with smaller id mean that they arrive and should be processed before the tuples with bigger id. Another way to identify the order of a tuple item is to separate the concept of application and system timestamp. In SECRET model [5], each stream item is composed of a tuple for event contents , an application timestamp, a system timestamp, and a batch-id value. The idea of batch-id is critical to SECRET system we do not mention in the thesis. In short, in practice, we assume that elements of a stream are totally ordered by the system timestamp and physical identifier.

In Apache Flink, for the flexibility, system accepts a user-defined timestamp function $f : \mathbb{TP} \rightarrow \mathbb{T}$ to map a tuple to its application timestamp value. One of the most common scenario is that the function f extract one attribute of Schema and consider it as timestamp value.

Consider the example of temperature sensors, the sensors feed a stream $S(TIME : long, TEMP : int)$ indicating that at the *TIME*, the temperature is *TEMP*. Since $TIME$ attribute has the *long* long, we are able to consider it as a timestamp due to function

$$f : (TIME, TEMP) \rightarrow SEC \tag{2.2}$$

However, we may receive the stream from a different timezone. Thus, the application timestamp must be converted to the current timezone for the sake of data integration. For instance, one acquires the timestamp value (in milliseconds) in next timeszone.

$$f : (TIME, TEMP) \rightarrow TIME + 3600 * 1000 \tag{2.3}$$

I propose the definition of Stream $s :< v >$ extends to $s :< v, t_{app}, t_{sys} >$, with $t_{app} = f(v)$, $t_{app} \in \mathbb{T}$, $t_{sys} \in \mathbb{T}$

For further analysis on execution model in Apache Flink, I propose a extend definition of a data stream as:

Definition 2.4 *A stream $\mathbb{S}$ is a countably infinite set of elements $s \in \mathbb{S}$. Each stream element $s :< v, t_{app}, t_{sys} >$, consists of a relational tuple v conforming to a schema TP, with an optional application time value $t_{app} = f(v) \in \mathbb{T}$, and a timestamp t_{sys} generated automatically by system, due to the event arrival.*

With the partial function f to extract application timestamp from tuple: $f : \mathbb{TP} \rightarrow \mathbb{T}$

Data Stream Properties

Data Stream may have the following properties [9]:

- They are considered as sequences of records, ordered by arrival time or by another ordered attributed such as generation time which is explicitly specified in schema, that arrive for processing over time instead of being available a priori. Totally order by time.

- They are emitted by a variety of external sources. Therefore, the system has no control over the arrival order or data rate, either within a stream or across multiple streams

- They are produced continually and, therefore, have unbounded, or at least unknown, length. Thus, a DSMS may not know if or when the stream "ends". We may set a time-out waiting for new event. Exceeding the time-out, the stream considerably ends.

- Typically, big volume of data arrives at very high speed so that data need to process on the fly. Once an element from a data stream model has been processed it is discarded or archived. Stream elements cannot be retrieved, unless it is explicitly stored in storage or memory, which typically is small relative to the size of the data stream. [4]

Stream Representations

Base Stream vs. Derived Stream

We distinguish 2 kinds of streams: *base stream*(source stream) and *derived stream*. Base stream stream is produced by the sources whereas derived stream is produced by continuous queries and their operators[3]. For example, [9] page 17 From now on, we give example queries on input stream named *StockTick*[Str] and the schema associated with the incoming tuples includes 4 fields:

- **Symbol**, a string field of maximum length 25 characters that contains the symbol for a stock being traded (e.g. IBM);
- **SourceTimestamp**, a timestamp field containing the time at which the tuple was generated by the source application(timestamp is represented with date time format or long integer);
- **Price**, a double field containing transaction price
- **Quantity**, a integer fields that contains the transaction volume
- **Exchange**, a string field of maximum length 4 that contains the name of Exchange the trade occurred on (e.g. NYSE)

A sample tuple represents the price of IBM stock unit is 81.37 at “1 May 2015 10:18:23" from NYSE market

```
<IBM,1430468303,81.37,NYSE>
```

The *StockTick* is emitted directly from source so that it is a base stream. However, a below *HighStockTick* stream is a derived stream originated from *StockTick*. *HighStockTick* contains only transaction of stock with price of more than $100 each unit.

```
CREATE STREAM HighStockTick AS
SELECT * FROM StockTick
WHERE price > 100
```

In practice, base stream are almost always append-only , mean that previously arrived stream elements are never modified. However, derived stream may or may not be append-only[9]. A derived stream that present the average transaction volumes between interval time $[t_1, t_2]$. The query produce an element s_1 to the stream immediately after t_2. However, an element of *StockTick* stream with timestamp $t_{12} \in [t_1, t_2]$ arrive late at $t_2 + \phi$. If the system takes into account of the late arrival element, it will update the previous average volumes at $[t_1, t_2]$. In this case, this derived stream is not append-only.

Note: another examples from **epl-guide** **Note**: stock example : http://www.codeproject.com/Articles/553
Introduction-to-Real-Time-Stock-Market-Data-Pro

Logical Stream vs. Physical Stream

Logical stream ? [8]Physical stream ? [8]

Logical stream is a conceptual and abstract data stream which is processed linearly through a series of chaining operators. According to logical data flow graph, one is able to observe the order of operators that data is processed and what are the input and output of process.

Physical stream flow graph indicates how system really process the data in data parallelism environment. Physical operator is replicated and the internal operator state is partitioned and segmented. Figure 2.1 depict the different between logical and physical data flow. A logical stream from source go straight through an aggregate and a filter operator before written to Sink, whereas physical streams are segmented and go through different internal replica of the same logical operator. Eventually, all physical streams will be merged and written to one Sink.

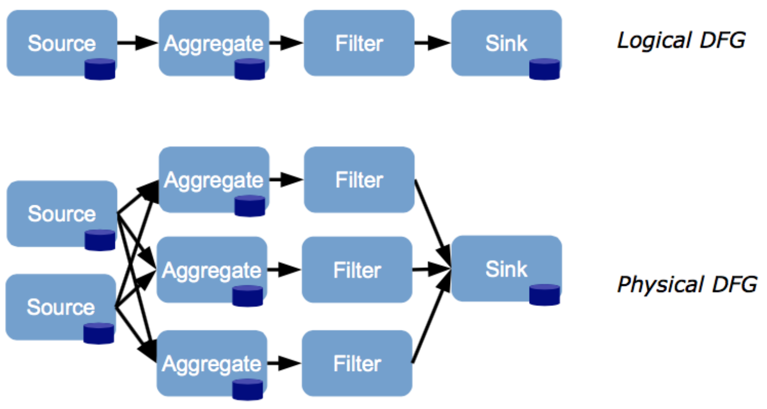

Figure 3: Extracting data parallelism from a logical DFG by replicating operators and segmenting their internal state in the corresponding physical DFG. The disk icon represents an operator's internal state.

Fig. 2.1 Stream Processing in Action. Henrique Andrade[Henrique Andrade and Turaga]

Signals[Option]

[9] Data streams often describe underlying “signals”. There are 4 ways in which a data stream can represent a signal

- aggregate model
- cash register model
- turnstile model
- reset model

2.2 Stream Window

Why do we need window?

It is often infeasible to maintain the entire history of the input data.

[2] a window over streaming data can be created, buffering individual tuples. Windows can be built according to a multitude of parameters that define what to buffer, resulting in many window variations. These variations differ in their policies with respect to evicting old data that should no longer be buffered, as well as in when to process the data that is already in the window.

Many stateful stream processing operators are designed to work on windows of tuples, making it a fundamental concept in stream processing. Therefore, a stream processing language must have rich windowing semantics to support the large diversity in how SPAs can consume data on a continuous basis.

Windows are defined with respect to a timestamp or sequence number attribute representing a tuple’s arrival time.

Window

[6] A **Window** W over a stream $\mathbb{S}$ is a finite subset of $\mathbb{S}$ [6]

A **Time-based Window** $W = [o, c)$ over a stream $\mathbb{S}$ os a finite subset of $\mathbb{S}$ containing all data elements $s \in \mathbb{S}$ where $o \leq s.t^{sys} < c$ [6]

Window size and slide [6]

Window specification

Paper : Window Specification over Data Stream

From the system's point of view, it is infeasible to store an entire stream. From the user's point of view, recently arrived data may be more useful. This motivates the use of windows to restrict the scope of continuous queries. Windows may be classified according the following criteria. Window may be classified according the following criteria:

2.2.1 Direction of movement

: fixed window, sliding window

Fixed starting and ending points define a fixed window. Moving starting and ending points (either forward or backward) creates a sliding window. The starting and ending points typically slide in the same direction, although it is possible for both to slide outwards, starting from a common position (creating an expanding window), or for both to slide inwards, starting from fixed, positions (creating a contracting window). One fixed point and one moving point define a landmark window. Usually, the starting point is fixed, and the ending point moves forward in time. There are a total of nine possibilities as the beginning and ending points may independently be fixed, moving forward or moving backward.

2.2.2 Definition of contents

time-based, count-based/tuple-based, partitioned windows, predicate window

Logical or time-based windows are defined in terms of a time interval, e.g., a time-based sliding window may maintain the last ten minutes of data. Physical (also known as count-based or tuple-based) windows are defined in terms of the number of tuples, e.g., a count-based sliding window may store the last 1000 tuples. When using count-based windows in queries with a GROUP BYcondition,it may be useful to maintain separate windows of equal size for each group, say, 1000 tuples each, rather than a single window of 1000 tuples; these are referred to as partitioned windows [Arasu et al., 2006]. The most general type is a predicate window [Ghanem et al., 2006], in which an arbitrary logical predicate (or SQL query) specifies the contents. Predicate windows are analogous to materialized views. For example, consider an on-line auction that produces three types of tuples for each item being sold: a "begin" tuple, zero or more "bid" tuples, followed by an "end" tuple that is generated when the item has been sold, presumably to the highest bidder. Assume that each tuple contains a timestamp, an item id, a type (being, bid or end), and other information such as the bid amount. A possible predicate window over this auction stream keeps track of all items that have not yet been sold. New items enter the window when their "begin" tuples appear; sold items "fall out" of the window when their "end" tuples appear.

2.2.3 Frequency of movement

jumping window, mixed jumping window, tumbling window

By default, a time-based window is updated at every time tick, and a count-based window is updated when a new tuple arrives. A jumping window is updated every k ticks or after every kth arrival. Note that a count-based window may be updated periodically, and a time-based window may be updated after some number of new tuples have arrived; these are referred to as mixed jumping windows [Ma et al., 2005]. If k is equal to the window size, then the result is a series of non-overlapping tumbling windows [Abadi et al., 2003]. In practice, tumbling windows, such as the one-minute windows in query Q1 from Sec- tion 2.1.1, are popular due to the simplicity of their implementation—at the end of each window, the query resets its state and starts over. Forward-sliding windows (time-based and count-based) are also appealing due to their intuitive semantics, especially with joins and aggregation, as we will discuss below. However, sliding windows are more difficult to implement than tumbling windows; over time, a continuous query must insert new tuples into a window and remove expired tuples that have fallen out of the window range.

(check Sliding Window Query Processing over Data Stream pages 25)

The notion of sliding windows requires at least an ordering on data stream elements. In many cases, the arrival orders of the elements suffices as an ïmplicit timestampättached to each data element. However, sometimes it is preferable to user ëxplicit timestamp̈provided as part of data stream. Formally, we say that a data stream consists of a set of (tuple, timestamp) pairs. Timestamp attribute could be a traditional timestamp or it could be a sequence number - all that is required is that it come from a totally ordered domain with a distance metric. The ordering induced by the timestamp is used when selecting the data elements making up a sliding window.

Notes: in CQL tuple-based sliding window may be non-deterministic - and therefore may not be appropriate - when timestamp are not unique

Notes: Streambase: tuple-at-a-time

Notes: Tuple relational calculus by Edgar F. Codd

Notes: http://en.wikipedia.org/wiki/Relational_algebra

Chapter 3

The execution semantic of Stream Processing in Flink (10 pages)

3.1 Heterogeneity

[6]

- Syntax
- Capability
- Execution Model

time-driven / tuple-driven / batch-driven

Order in stream

total order vs partial order

Chapter 4

Continuous Query Semantics and Operators (10 pages)

- Window alone come with Trigger
    - Window + Every:

    + Window : Eviction

    + Every: Trigger

    Eviction: number of events to keep (start of window)

    Trigger : when to start firing the function (end of window)

    startTime: end of the first Window

Chapter 5

Implementations

5.1 Reasonably long section title

5.1.1 This is a short title

I’m going to randomly include a picture Figure 5.1.

If you have trouble viewing this document contact Krishna at: kks32@cam.ac.uk or raise an issue at https://github.com/kks32/phd-thesis-template/

Enumeration

1. The first topic is dull
2. The second topic is duller
   (a) The first subtopic is silly
   (b) The second subtopic is stupid
3. The third topic is the dullest

itemize

- The first topic is dull
- The second topic is duller
  - The first subtopic is silly

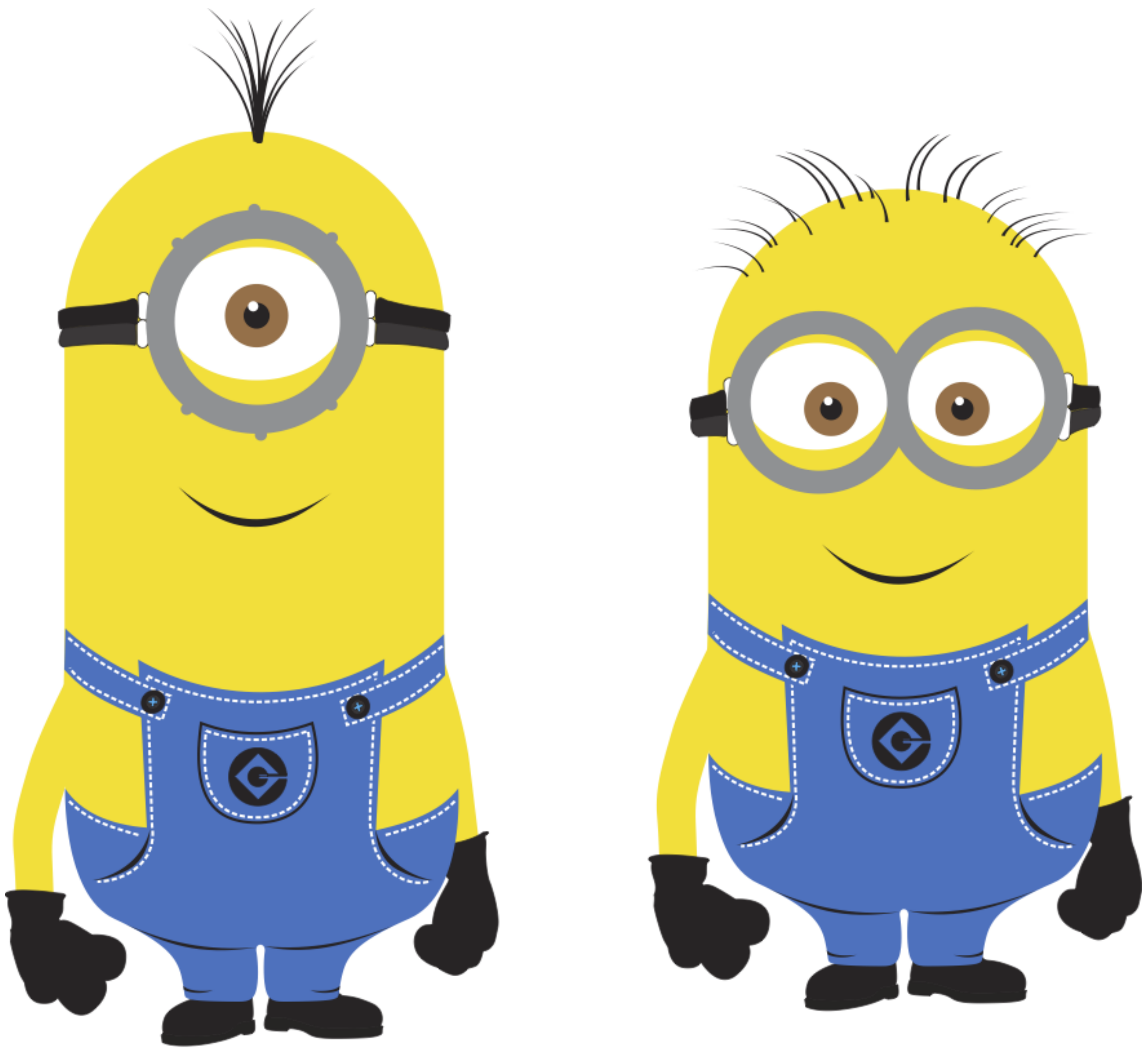

Fig. 5.1 This is just a long figure caption for the minion in Despicable Me from Pixar

    - The second subtopic is stupid
- The third topic is the dullest

description

The first topic is dull

The second topic is duller

> **The first subtopic** is silly
>
> **The second subtopic** is stupid

The third topic is the dullest

5.2 Hidden section

Lorem ipsum dolor sit amet, *consectetur adipiscing elit.* In magna nisi, aliquam id blandit id, congue ac est. Fusce porta consequat leo. Proin feugiat at felis vel consectetur. Ut tempus ipsum sit amet congue posuere. Nulla varius rutrum quam. Donec sed purus luctus, faucibus velit id, ultrices sapien. Cras diam purus, tincidunt eget tristique ut, egestas quis nulla. Curabitur vel iaculis lectus. Nunc nulla urna, ultrices et eleifend in, accumsan ut erat. In ut ante leo. Aenean a lacinia nisl, sit amet ullamcorper dolor. Maecenas blandit, tortor ut scelerisque congue, velit diam volutpat metus, sed vestibulum eros justo ut nulla. Etiam nec ipsum non enim luctus porta in in massa. Cras arcu urna, malesuada ut tellus ut, pellentesque mollis risus.Morbi vel tortor imperdiet arcu auctor mattis sit amet eu nisi. Nulla gravida urna vel nisl egestas varius. Aliquam posuere ante quis malesuada dignissim. Mauris ultrices tristique eros, a dignissim nisl iaculis nec. Praesent dapibus tincidunt mauris nec tempor. Curabitur et consequat nisi. Quisque viverra egestas risus, ut sodales enim blandit at. Mauris quis odio nulla. Cras euismod turpis magna, in facilisis diam congue non. Mauris faucibus nisl a orci dictum, et tempus mi cursus.

Etiam elementum tristique lacus, sit amet eleifend nibh eleifend sed [1]. Maecenas dapibu augue ut urna malesuada, non tempor nibh mollis. Donec sed sem sollicitudin, convallis velit aliquam, tincidunt diam. In eu venenatis lorem. Aliquam non augue porttitor tellus faucibus porta et nec ante. Proin sodales, libero vitae commodo sodales, dolor nisi cursus magna, non tincidunt ipsum nibh eget purus. Nam rutrum tincidunt arcu, tincidunt vulputate mi sagittis id. Proin et nisi nec orci tincidunt auctor et porta elit. Praesent eu dolor ac magna cursus euismod. Integer non dictum nunc. 5.2c

[1]My footnote goes blah blah blah! ...

(a) Tom and Jerry

(b) Wall-E

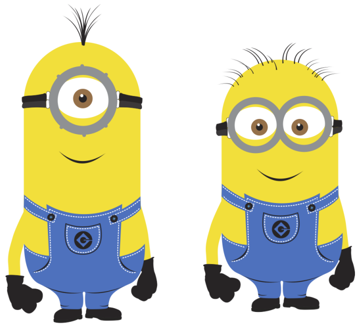
(c) Minions

Fig. 5.2 Best Animations

Subplots

I can cite Wall-E (see Fig. 5.2b) and Minions in despicable me (Fig. 5.2c) or I can cite the whole figure as Fig. 5.2

References

[Str] Streamsql tutorial. http://www.streambase.com/developers/docs/latest/streamsql/usingstreamsql.html/. Accessed May 10, 2015.

[2] Andrade, H. C. M., Gedik, B., and Turaga, D. S. (2014). *Fundamentals of Stream Processing*. Cambridge University Press. Cambridge Books Online.

[3] Arasu, A., Babu, S., and Widom, J. (2006). The cql continuous query language: Semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142.

[4] Babcock, B., Babu, S., Datar, M., Motwani, R., and Widom, J. (2002). Models and issues in data stream systems. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, pages 1–16, New York, NY, USA. ACM.

[5] Botan, I., Derakhshan, R., Dindar, N., Haas, L., Miller, R. J., and Tatbul, N. (2010). Secret: A model for analysis of the execution semantics of stream processing systems. *Proc. VLDB Endow.*, 3(1-2):232–243.

[6] Dindar, N., Tatbul, N., Miller, R. J., Haas, L. M., and Botan, I. (2013). Modeling the execution semantics of stream processing engines with secret. *The VLDB Journal*, 22(4):421–446.

[Henrique Andrade and Turaga] Henrique Andrade, B. G. and Turaga, D. Stream processing in action.

[8] Krämer, J. and Seeger, B. (2009). Semantics and implementation of continuous sliding window queries over data streams. *ACM Trans. Database Syst.*, 34(1):4:1–4:49.

[9] Lukasz Golab, M. T. O. (2010). Data stream management.

[10] Petit, L., Labbé, C., and Roncancio, C. L. (2010). An algebric window model for data stream management. In *Proceedings of the Ninth ACM International Workshop on Data Engineering for Wireless and Mobile Access*, MobiDE '10, pages 17–24, New York, NY, USA. ACM.

[11] Petit, L., Labbé, C., and Roncancio, C. L. (2012). Revisiting formal ordering in data stream querying. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 813–818, New York, NY, USA. ACM.

[Takada] Takada, M. *Distributed Systems For Fun and Profit*.