Part I

Fundamentals

1 What brought us here?

1.1 Overview

The world has become information-driven, with many facets of business and government being fully automated and their systems being instrumented and interconnected. On the one hand, private and public organizations have been investing heavily in deploying sensors and infrastructure to collect readings from these sensors, on a continuous basis. On the other hand, the need to monitor and act on information from the sensors in the field to drive rapid decisions, to tweak production processes, to tweak logistics choices, and, ultimately, to better monitor and manage physical systems, is now fundamental to many organizations.

The emergence of stream processing was driven by increasingly stringent data management, processing, and analysis needs from business and scientific applications, coupled with the confluence of two major technological and scientific shifts: first, the advances in software and hardware technologies for database, data management, and distributed systems, and, second, the advances in supporting techniques in signal processing, statistics, data mining, and in optimization theory.

In Section 1.2, we will look more deeply into the data processing requirements that led to the design of stream processing systems and applications. In Section 1.3, we will trace the roots of the theoretical and engineering underpinnings that enabled these applications, as well as the middleware supporting them. While providing this historical perspective, we will illustrate how stream processing uses and extends these fundamental building blocks. We will close this chapter with Section 1.4 where we describe how all these different technologies align to form the backbone of stream processing software.

1.2 Towards continuous data processing: the requirements

In 2010, Walmart, a retailer, was reported to handle more than 1 million customer transactions every hour [1]. According to OCC, an equity derivatives clearing organization, average daily volumes of options contracts[1] by October 2012 were around 14 million [2]. The New York Stock Exchange (NYSE), the world's largest stock exchange

[1] An *option* is a derivative financial instrument specifying a contract for a future transaction on an asset at a reference price, for instance, the right to buy a share of IBM stock for $300 on February 1, 2016.

by market capitalization of its listed companies, traded more than 800 million shares on a typical day in October 2012 [3]. In other business domains, IBM in a comprehensive report on data management and processing challenges [4] pointed out that: (1) by the end of 2011 there were about 30 billion Radio-Frequency IDentification (RFID) tags in circulation, each one a potential data source; (2) T-Mobile, a phone company, processed more than 17 billion events, including phone calls, text messages, and data traffic over its networks; and (3) the avionics of a modern commercial jet outputs 10 terabytes (TB) of data, per engine, during every half hour of operation. These examples are just a few illustrations of the data processing challenges organizations must face today, where all of this data must be ingested, processed, and analyzed quickly, potentially as it is being continuously produced.

Indeed, as the world gets more connected and instrumented, there is a deluge of digital data coming from various software and hardware sensors in the form of continuous streams of data [5]. Examples can be found in several domains ranging from financial markets [6], multi-modal surveillance and law enforcement [7, 8], manufacturing [6], healthcare [9, 10], interconnected city infrastructure and traffic systems [11, 12], large-scale infrastructure monitoring [8], to scientific and environmental domains such as radio astronomy [13, 14] and water management [15, 16].

In all of these domains there is an increasing need to gather, process, and analyze live streams of data to extract insights in real-time and to detect emerging patterns and outliers. While the requirements are very clear, there are several challenges that must be overcome for this to be possible. We will discuss some of these challenges in the rest of this section, and provide a description of the techniques and software development strategies that can be used to address them in the rest of this book.

Processing large quantities of distributed data

The fundamental goal of stream processing is to process live data in a fully integrated fashion, providing real-time information and results to consumers and end-users, while monitoring and aggregating new information for supporting medium- and long-term decision-making.

The high volume of streaming data flowing from distributed sources often makes it impossible to use the storage- and Input/Output (I/O)-centric relational database and data warehousing model, where *all* data is first stored on disk and later retrieved for processing and analysis. Additionally, continuous monitoring across *large*, potentially multi-site applications that connect remote sensing and processing, needs to rely on a distributed computing infrastructure. The real-time monitoring, analysis, and control nature of these applications also implies the need to adapt to dynamic characteristics of their workloads and to instantaneous resource availability of the systems.

Addressing stringent latency and bandwidth constraints

Stream Processing Applications (SPAs) need to satisfy several performance requirements in terms of latency and throughput. Specifically, the data processing must keep up with data ingest rates, while providing high quality analytical results as quickly as possible.

Consider, for example, certain types of finance engineering applications. Increasingly, large amounts of market data must be processed with very low latencies to determine trading opportunities [17]. These requirements translate into performance constraints in terms of data decoding, demultiplexing, and delivery between different components in an application, where peak rates of millions of messages per second and sub-millisecond end-to-end latency are very common.

Processing heterogeneous data

The raw data streams analyzed by SPAs are often heterogeneous in format, content, rates, and noise levels, making generic data processing and analysis difficult without appropriate pre-processing steps. Data streams may also consist of unstructured data types such as audio, video, image, and text that cannot easily be handled using a traditional data management infrastructure. The fundamental issues here range from ingesting the data from a disparate collection of sensors with different protocols and data formats, to cleaning, normalization, and alignment so the raw data is adequately prepared for in-depth processing.

Applying complex analytics to streaming data

The types of analysis required by typical SPAs range from simple monitoring and pattern detection, to more complex *data exploration* tasks, where new information must be distilled from raw data in an automated fashion. Complex analysis is usually needed in applications that are part of sophisticated Decision Support Systems (DSSs).

The set of relevant data sources for stream data exploration applications is time-varying, typically large, and potentially unbounded in contrast with the finite computational cycles available. Thus, decisions such as which sources to ingest data from and what fraction of data per source to consider must be autonomic and driven by findings in the data, which are periodically fed back to the earlier data processing stages of an application.

Hence, the analytics employed by a data exploration application must be adaptive, to handle the dynamic and potentially unbounded set of available data streams. There is also a need for both *supervised* and *unsupervised analysis*. The constraint on the available computational resources requires using *approximation algorithms* [18] that explicitly trade off accuracy for complexity, making the appropriate choices given the instantaneous conditions of processing resources and workload.

These applications also often need to support *hypothesis-based analysis* [19], where the processing is dynamically changed based on the results being produced. In other words, based on validated (or discarded) hypotheses, different sources, different processing, and different analytic algorithms may need to be switched on and off as the application runs. Finally, these applications are also mostly "best-effort" in nature, attempting to do as best as possible to identify useful information with the resources that are available.

Traditionally, these analytic tasks have been implemented with data exploration and modeling tools such as spreadsheets and mathematical workbenches, to analyze medium-size stored datasets. Stream processing should be used to supplement

traditional tools by providing continuous and early access to relevant information, such that decisions can be made more expeditiously. Distributed stream processing also enables these applications to be scaled up, allowing more analysis to be performed at wire speeds.

Providing long-term high availability

The long-running and continuous nature of SPAs requires the construction and maintenance of state information that may include analytical models, operating parameters, data summaries, and performance metric snapshots. Thus, it is critical that applications have mechanisms to maintain this internal state appropriately without letting it grow unboundedly, which can lead to performance degradation and potential failures. Clearly, these applications are uniquely prone to suffering from these problems due the continuous nature of the processing.

Additionally, there is a need for tolerance to failures in application components or data sources, as well as failures in the processing hardware and middleware infrastructure. For these reasons, applications must be designed with *fault tolerance* and resilience requirements in mind.

An important aspect in terms of fault tolerance is that different segments of a SPA may require different levels of reliability. For instance, tolerance to sporadic data loss is, in many cases, acceptable for certain parts of an application, as long as the amount of error can be bounded and the accuracy of the results can be properly assessed. Other application segments, however, cannot tolerate any failures as they may contain a critical persistent state that must survive even catastrophic failures. Hence, the application design needs to consider the use of *partial fault tolerance* techniques [20, 21] when appropriate.

Finally, because of the additional overhead imposed by fault tolerance, taking into account *where* and *how* fault-tolerant capabilities should be deployed can make a substantial impact on minimizing computational and capital costs.

Distilling and presenting actionable information

Enabling faster information analysis and discovery is only one part of designing and deploying a new SPA. Deciding what data to keep for further analysis, what results to present, how to rank these results, and, finally, how to present them in a *consumable* form, on a continuous basis, are equally important.

Tightening the loop from data collection to result generation creates additional challenges in presenting freshly computed information. Such challenges also come with additional opportunities to increase the interactivity and relevance of the information being displayed [22].

1.3 Stream processing foundations

The development of Stream Processing Systems (SPSs) was the result of the constant evolution over the past 50 years in the technology used to store, organize, and analyze the increasingly larger amounts of information that organizations must handle.

These trends have also enabled the development of modern relational databases and, later, data warehousing technologies, as well as object-oriented databases and column-oriented databases.

More recently, Google and Yahoo have made the *MapReduce* framework popular for processing certain types of large-scale data aggregation/summarization tasks. This development builds upon several data processing techniques proposed by the research community [23, 24, 25, 26]. Additionally, multiple technologies have been developed to provide historical overviews and data summarization over long periods of time [25, 27, 28, 29]. Some of these technologies have been commercialized as part of data warehousing products as well as Business Intelligence (BI) platforms. We discuss all of these technologies in more detail in Section 1.3.1.

The need for tackling larger processing workloads has led to the development of *distributed systems* with the capability of dividing up a workload across processors. This area has been a particularly fertile ground for hardware and software experimentation, dating back to the beginning of the modern computer era in the early 1950s. The field has produced a large variety of platforms that can be found commercially today, ranging from simple multicore processors, to clusters, to specialized acceleration hardware, and supercomputers. Despite the astounding hardware improvements seen over the last few decades, substantial software engineering challenges still exist today in terms of how to effectively write software for these platforms and, more importantly, to ensure that the software can evolve and benefit from architectural improvements. We discuss these topics in more detail in Section 1.3.2.

The need for large-scale data analysis has led to several advances in *signal processing* and *statistics*. Several signal processing techniques for pre-processing, cleaning, denoising, tracking, and forecasting for streaming data have been developed over the last few decades. Statistical techniques for data summarization through *descriptive statistics*, *data fitting*, *hypothesis testing*, and *outlier detection* have also been extensively investigated with new methods frequently emerging from researchers in these fields.

More recently, researchers in statistics and computer science have coined the term *data mining* as a way to refer to techniques concerned primarily with extracting patterns from data in the form of models represented as *association rules*, *classifier models*, *regression models*, and *cluster models*. Of particular interest are the advancements that have been made on continuous (or online) analytical methods, which allow the incremental extraction of the data characteristics and incremental updates to these models. We discuss these topics in more detail in Section 1.3.3.

Finally, all of these developments have been supported by advances in *optimization theory*. These techniques make it possible to obtain optimal or *good enough* solutions in many areas of data analysis and resource management. While the roots of optimization theory can be traced as far back as Gauss' *steepest descent method* [30], modern optimization methods such as *linear programming* and *quadratic programming* techniques have only been developed over the last few decades. We discuss these topics in more detail in Section 1.3.4.

1.3.1 Data management technologies

The most prevalent data management technology in use nowadays is the *relational* database. Data management applications require transaction management, resiliency, and access controls. All these requirements and associated problems have been progressively studied by the research community, and solutions have been implemented by open-source and commercially available database and other data management systems. The long history behind data management is deeply intertwined with the development of computer science.

In the late 1960s and early 1970s, commercial data management relied on mainly on Data Base Management Systems (DBMSs) implementing either the *network model* [31] or the *hierarchical model* [32]. In this context, a *model* describes how the information in the database system is organized internally.

The network model, defined originally by the Codasyl Data Base Task Group [33], represented information in terms of a *graph* in which objects (e.g., an `employee` record) are *nodes* and relationships (e.g., an employee working in a department is represented by a relationship between two objects: a specific `employee` object and a specific `department` object) are *arcs*. The network model was tightly coupled with the Cobol language [34], which was used for writing applications that queried and manipulated the data stored in a database. GE's ID database system [35], relied on this model and was released in 1964.

The hierarchical model structured information using two basic concepts: records (e.g., a structure with attributes representing an `employee` and another representing a `department`) and parent–child relationships (PCR) (e.g., a `department/employee` PCR). Note that where the hierarchical model structured data as a tree of records, with each record having one parent record and potentially many children, the network model allowed each record to have multiple parent and child records, organized as a graph structure. Hierarchical model-based database management systems first became available with IBM's IMS/VS in the late 1960s. Many of these systems are still around and can be programmed using a plethora of host languages, from Cobol, to Fortran [36] and PL/1 [37].

In the early 1970s, the term *relational model* was coined by Codd [38]. The relational model sits atop a theoretical framework referred to as relational algebra, where the data is represented by *relations*. Some of the seminal relational data processing, including the original implementation of its SQL query language, came from IBM's System R implementation [39, 40] as well as from other efforts that were taking place in academia [41].

The theory and technological underpinnings behind relational databases ushered an enormous amount of evolution in data management, due to the simplicity, elegance, and expressiveness embodied in the relational model.

Relational model: fundamental concepts

In relational databases, the data is primarily organized in *relations* or *tables*, which comprise a set of *tuples* that have the same *schema*, or set of attributes (e.g., a `transaction identifier`, a `client identifier`, a `ticker symbol`, for a hypothetical relation representing a stock market transaction) [42].

From a mathematical standpoint, a *relation* is any subset of the Cartesian product of one or more domains. A *tuple* is simply a member of a relation. The collection of relation schemas used to represent information is called a relational *database schema*, and *relationships* can be established between the different relations.

From this brief description it can be seen that relational databases rely on three fundamental components. First, the Entity Relationship (ER) model, which depicts data management schemas in the form of a graph diagram. One such diagram represents *entity* sets (e.g., the set of `employees`), entity *attributes* (e.g., an employee's `name`), and *relationships* (e.g., an employee works for a `department`, i.e., another entity).

Second, the relational algebra, which defines operators for manipulating relations. Arguably, the most important breakthrough of the relational algebra has been the conceptual and intuitive nature of these operators: the *union* $R \cup S$ of relations R and S results in the set of tuples that are in either relation; the *set difference* $R - S$ results in the set of tuples in R and not in S; the *Cartesian product* $R \times S$ results in the set of every tuple in R combined with every tuple in S, each resulting tuple including the attributes from both original tuples; the *projection* $\pi_{a_1,a_2,\ldots,a_n}(R)$ is the operation whereby tuples from a relation R are stripped of some of the attributes or have their attributes rearranged, resulting in attributes $a_1, a_2, \ldots, a_n$ being output; and, finally, the *selection* $\sigma_F(R)$ operation employs a formula F, comprising operands such as constants or attribute names combined with arithmetic comparison operators as well as logical operators, and computes the set of tuples resulting from applying the function F to each tuple that belongs to relation R. Numerous other operations can also be formulated in terms of these basic operators.

Third, the Structured Query Language (SQL), formally specified by an ISO/IEC standard [43], which is a relational database management system's main user interface. The SQL concepts *table*, *row*, and *column* map directly to counterparts in relational algebra, *relation*, *tuple*, and *attribute*, respectively. Likewise, all the relational algebra operators can be translated directly into SQL constructs. SQL also includes constructs for data and schema definition (effectively working as both a Data Definition Language (DDL) and as a Data Manipulation Language (DML)), as well as constructs for querying, updating, deleting, and carrying out data aggregation tasks [44].

Data warehousing

A data warehouse is a large-scale data repository and the associated system supporting it [31]. A warehouse is different from a regular database in the sense that it is primarily intended as the processing engine behind decision-support applications, where transactional support (also referred to as Online Transaction Processing (OLTP)) and *fine-grain* query capabilities are of lesser importance.

Furthermore, a data warehouse typically has a data acquisition component responsible for fetching and pre-processing the input data that it will eventually store. This batch processing nature leads to hourly, daily, and sometimes longer update cycles, which require drawing data from different database sources and cleaning, normalizing, and reformatting it. This mode of operation contrasts with regular databases where data updates are incremental, resulting from the query workload imposed by an application.

The types of applications supported by warehouses, namely Online Analytical Processing (OLAP), DSS, and data mining, operate on aggregated data. Hence, efficient bulk data retrieval, processing, cleaning, and cross-database integration, coupled with extensive aggregation and data presentation capabilities, are the hallmark of data warehousing technologies.

Also in contrast with relational databases, typical operations include *roll-ups* (i.e., summarizing data with a progressive amount of generalization, for example, to produce weekly sales report from daily sales data), *drill-downs* (i.e., breaking down a summarization with a decreased amount of generalization), and *slicing* and *dicing* (i.e., projecting out different data dimensions from a multi-dimensional, aggregation data product such as a *data cube*[2]) [45].

While the basic data operations possible within a data warehouse can provide multiple perspectives on large quantities of data, they often lack capabilities for automated knowledge discovery. Hence, a natural evolution in data warehousing technology involves integrating *data mining* capabilities as part of the data warehouse toolset. Not surprisingly, products such as IBM InfoSphere Data Warehouse [46], IBM SPSS Modeler [47], Oracle Data Mining [48], Teradata Warehouse Miner [49], among others, provide these capabilities. Data mining is further discussed in Section 1.3.3.

Business intelligence frameworks

Building on data warehousing technologies, several BI middleware technologies have sprung up. Their primary focus is on building visualization *dashboards* for depicting up-to-date data and results. As is the case with stream processing, these technologies aim at providing actionable intelligence to decision makers, as well as predictive capabilities based on trends observed in historical data. Nevertheless, the technological focus in this case is less on data management and more on assembling and post-processing results by relying on other systems as data sources.

In many ways, modern business intelligence frameworks such as those provided by IBM (IBM Cognos Now! [50]), Oracle (Oracle Hyperion [51]), MicroStrategy [52], draw on basic ideas from data warehousing. In many cases, they integrate with data warehouses themselves. Most of the added benefits include providing (1) additional operators to post-process the data coming from warehouses, from data mining middleware, as well as from regular DBMSs, (2) continuous processing and exception handling and notification for addressing abnormal conditions arising from newly arrived data, and (3) toolkits to produce interactive data visualization interfaces organized in executive dashboards.

Emerging data management technologies

While relational databases are currently dominant in the marketplace, several technologies have been developed over the last two decades to address certain limitations in relational DBMSs or, simply, to complement them.

[2] A data cube is a data structure designed for speeding up the analysis of data using multiple aggregation perspectives. It is usually organized as a multi-dimensional matrix [45].