

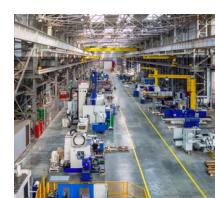
Introduction to Stream Processing

Outline

- Motivating Applications
 - Need for Stream Processing
 - Classes of Applications
 - Key Application Requirements
- Emergence of Stream Processing
 - Databases → Rule Engines → Stream Processing
- Stream Processing Systems
 - Examples
 - Anatomy of a Stream Processing System

Data Deluge

Every natural system and man-made system
is becoming interconnected, instrumented and intelligent



Utilities

Healthcare

Cities

Food

Manufacturing

Transportation

Oil & Gas

Safety

User Data:
Smart Phones
and Homes



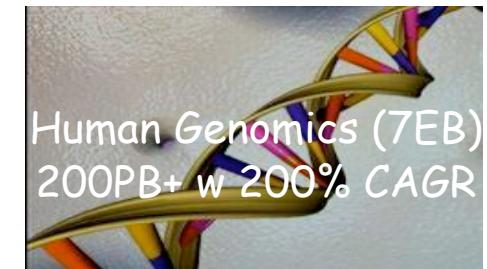
Data Deluge



You Tube
500 hrs of
videos/min, 2x in
5 months

95 Million Images/Day

FACEBOOK
30 Billion Pieces
Content/month



Total digital data created in 2012 1.8 Zettabytes. In 2017 16.3, in 2020 44 ZettaBytes! Not just numeric timeseries, mostly unstructured

Need for Streaming Analysis

- Ever increasing
 - Number of data sources
 - Volume of data
 - Variety of data
 - Velocity of data
- To utilize data
 - React fast – low latency
 - Analyze more data – high volume
 - Correlate different sources – variety
 - Apply complex, predictive analytics, and learning
 - Present live results and take action

Healthcare and Internet of Things



MiniMed Connect



- ④ Integrate with other context information from phone
- ① Capture streaming data, and 24 hour snapshot
- ② Pre-process, clean, transform and prepare for storage and summary
- ③ Provide authenticated access and summaries of streaming data, customized to end-user
- ⑥ Apply models against streaming data to create real-time insights and provide the user with: Insulin Recommendations, Behavior Recommendations, and identified deviations and patterns



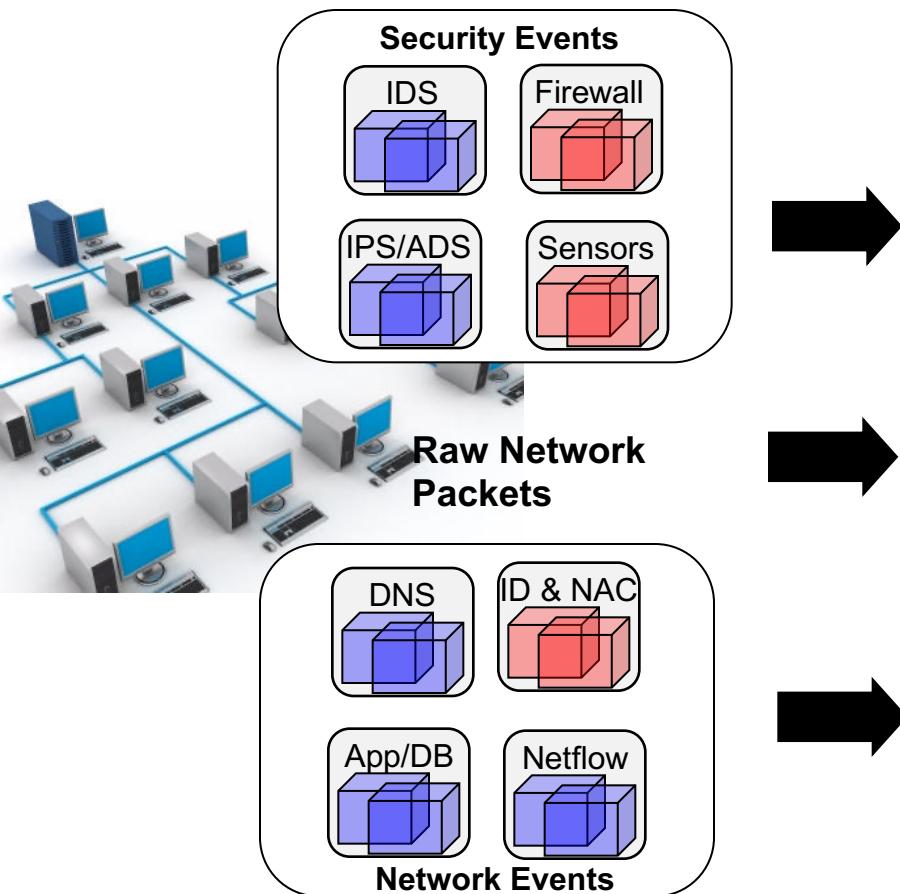
Cloud



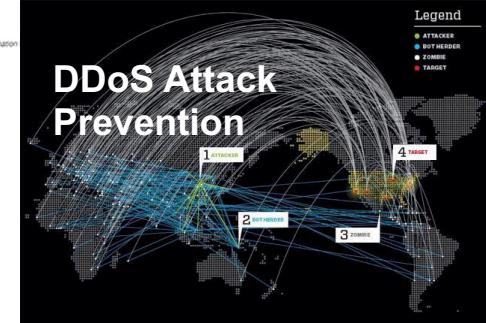
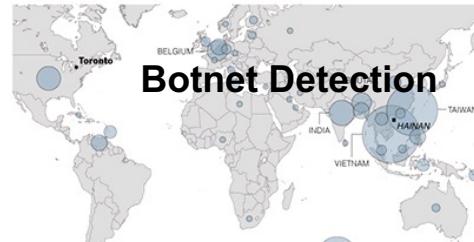
A Streaming Healthcare Application

- Streaming Timeseries Data
 - Heterogeneous (Health, Context, Location)
 - Multiple sources
 - Noise, Delay, Missing values, Unobserved Information
- Streaming Analysis
 - Continuous prediction, pattern mining, live alerting (hypo prediction)
 - Online learning
- Systems
 - Scaling, distributed processing, fault tolerance

Use Case: Cybersecurity



The Vast Reach of 'GhostNet'
Researchers have detected an intelligence gathering operation involving at least 1,295 compromised computers. Below, the locations of 347 of the compromised machines, many of which were tracked to diplomatic and economic government offices of South and Southeast Asian countries.



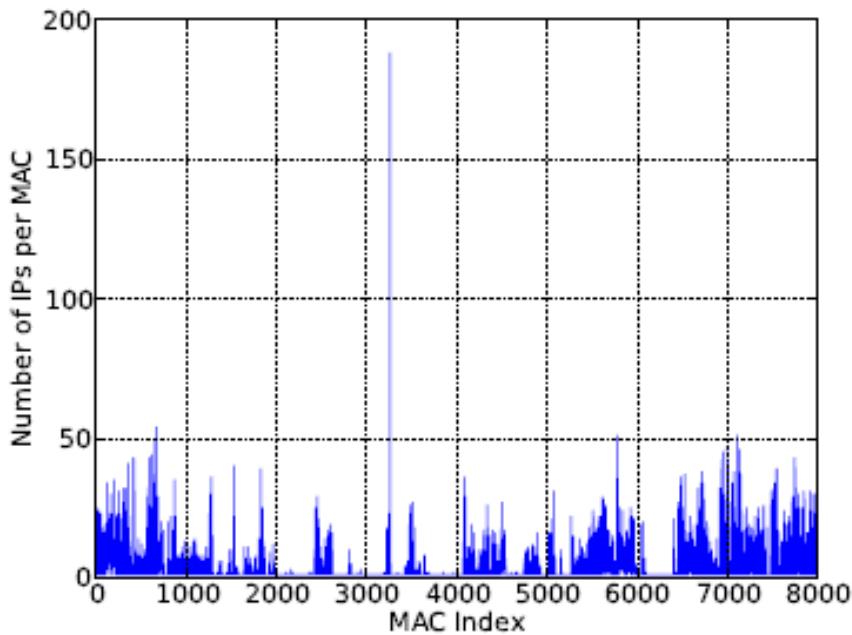
Automated Threat Analysis and Mitigation
Discriminative network behavioral patterns for infection detection & prevention

Securing assets and information

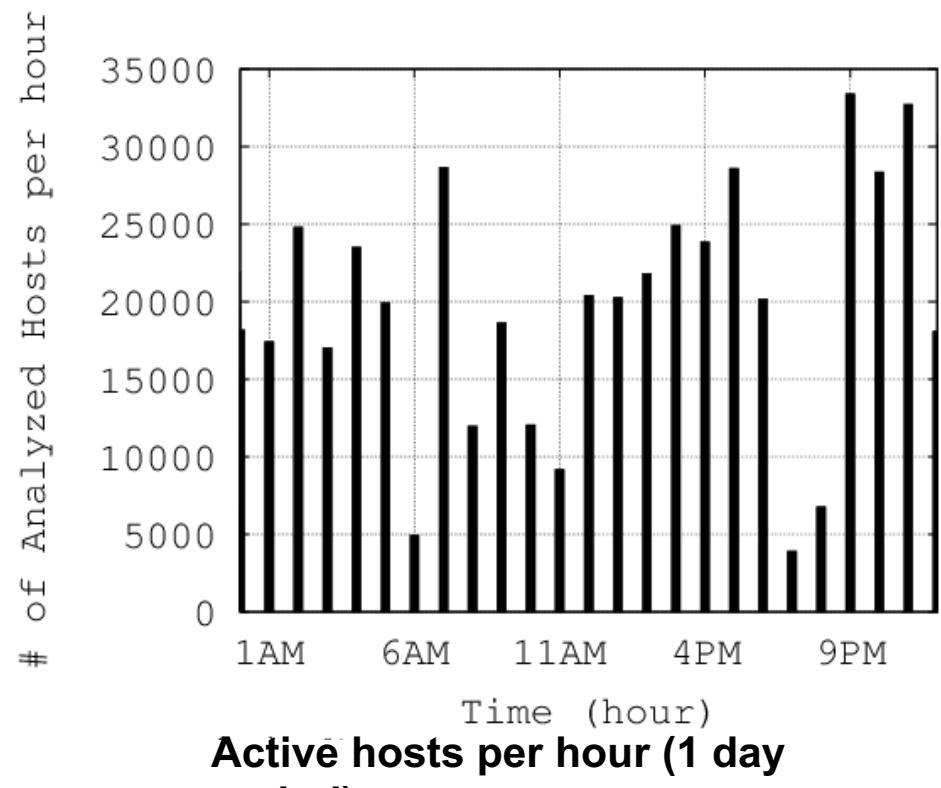
A Streaming Network Security Application

- Streaming Timeseries Data
 - Heterogeneous (Control Information, Raw Packets)
 - High throughput
 - Structured and Unstructured Content
- Streaming Analysis
 - Continuous threat monitoring, outlier detection
 - Online and continuous learning
- Systems
 - Scaling, distributed processing, fault tolerance

Network Dynamics

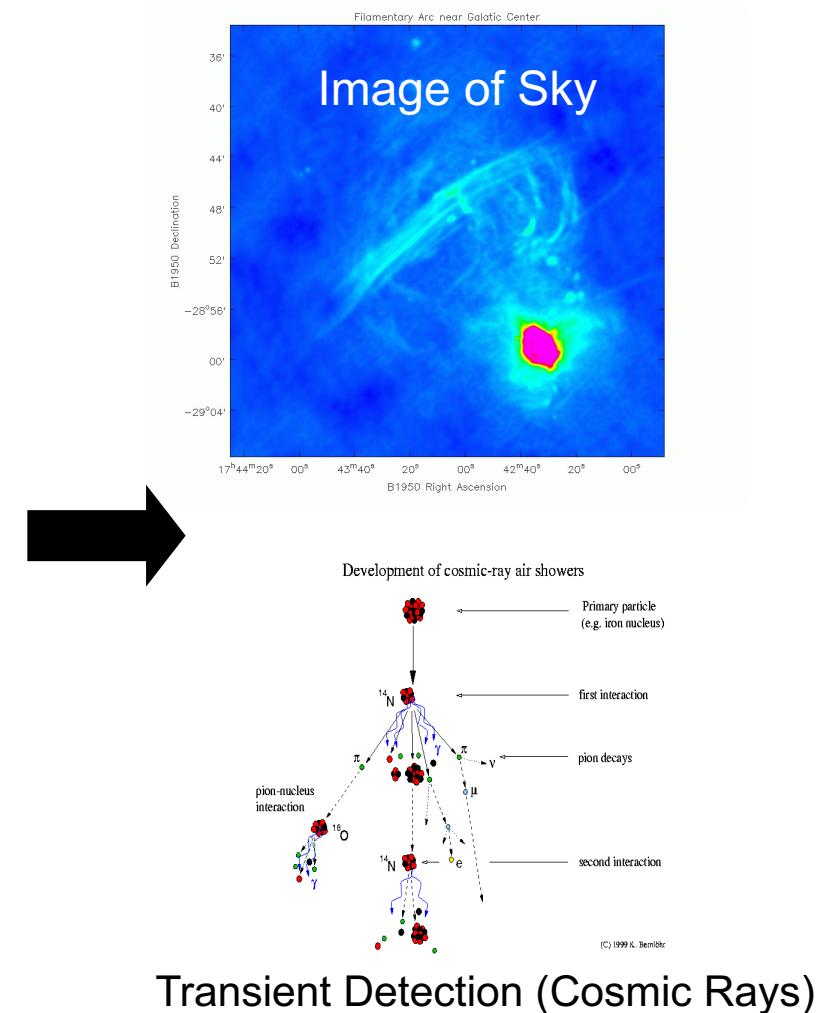
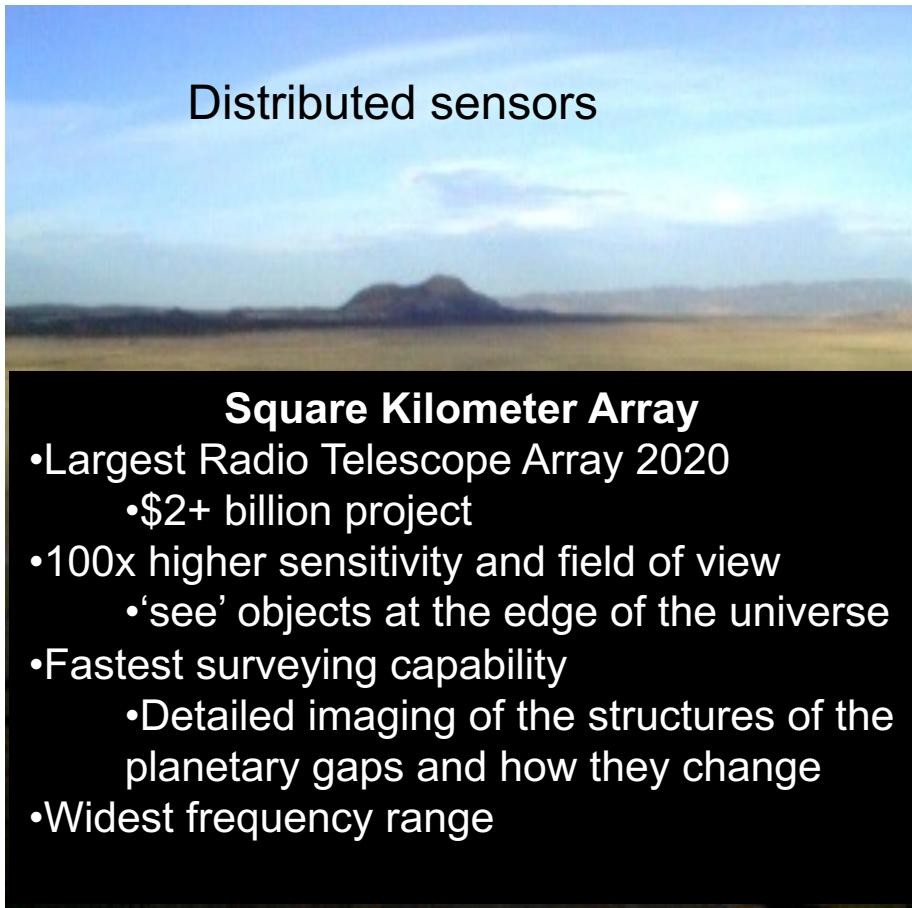


DHCP Churn: Dynamic Client Identity



Active hosts per hour (1 day period)

Use Case: Radio Astronomy



Several Other Use Cases



Streaming Application Requirements

- Streaming Data
 - Wide Range of Data Rates: Manufacturing: 5-10 Mbps, Astronomy: ~x00 Gbps, Healthcare: ~x00 Kbps per patient
- Streaming Analysis
 - Hierarchical, open-ended, and long running analysis. Online learning
- Multimodal Data Analysis
 - Correlated primary sources, Structured or Unstructured
- Distributed Analysis
 - Decomposable into flowgraphs, distributed data, processing
- High Performance
 - Real-time, Low-latency, high throughput, scaling
- Dynamic and Adaptive Analysis
 - Source, data, analysis and resource variability
- Loss Tolerant Analysis
 - Highly noisy and lossy data, graceful degradation under failure

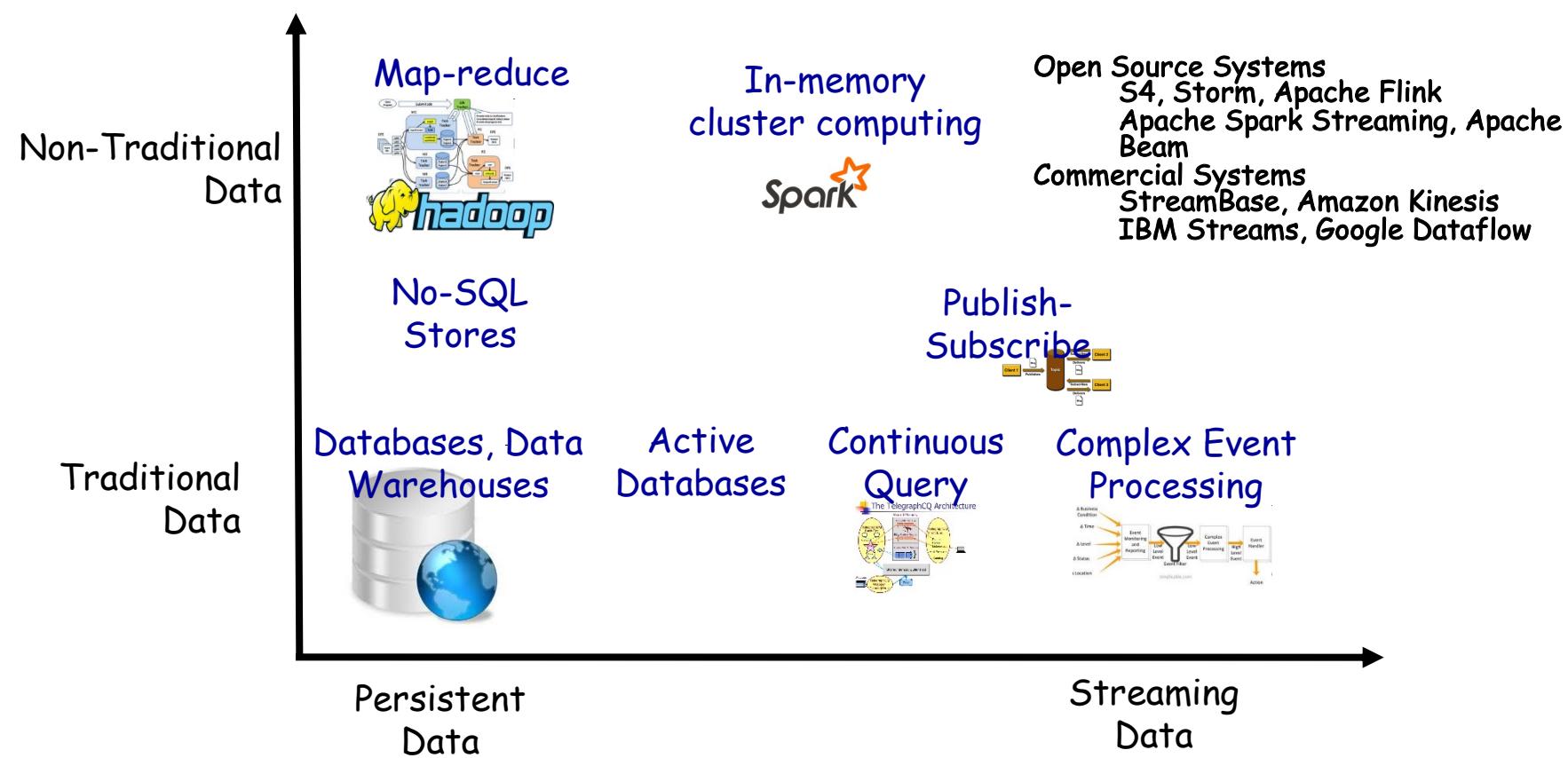
Classes of Stream Processing Apps

- Performance Driven
- Control and Decision Support
- Exploration Driven
- Historical Analysis and Simulation

Why Stream Processing Systems?

- Need a natural paradigm for handling streaming data applications
- Programming streaming applications
 - Language, constructs
 - IDE, Debugging, Visualization
- Running and scaling streaming applications
 - Runtime spanning distributed machines
- Tools
 - Toolkits, connectors, adapters, optimizers
- Makes it “easy” to develop, deploy, and manage streaming applications

Towards Stream Processing Systems



Towards Stream Processing Systems

System	Streaming Analysis	Distributed Processing	High Performance and Scalability	Unstructured Analysis	Fault Tolerance	Adaptive Analysis
Databases (DB2, Oracle, MySQL)	No	Yes	Partly	No	Yes	No
Parallel Processing (PVM, MPI, OpenMP)	No	Yes	Yes	Yes	Yes	No
Active Databases (Ode, HiPac, Samos)	Partly	Partly	No	No	Yes	Yes
Continuous Query Systems (NiagaraCQ, OpenCQ)	Partly	Partly	No	No	Yes	Yes
Pub-Sub Systems (Gryphon, Siena, Padres)	Yes	Yes	No	No	Yes	Partly
CEP Systems (Esper, SASE, IBM WBE, Tibco BE, Oracle CEP)	Yes	Partly	Partly	No	Yes	Partly
Map-Reduce (Hadoop)	No	Yes	Yes	Yes	Yes	No

Evolution towards Stream Processing Systems

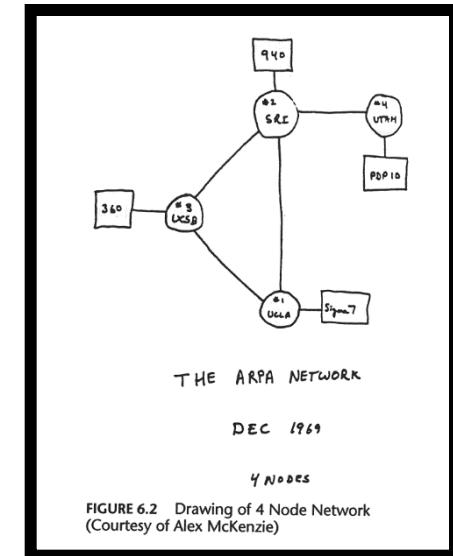
- Distributed Computing Systems
- Data Management Systems
 - Databases and Data Warehouse
- Information Flow Processing Systems
 - Active Databases
 - Continuous Query Systems
 - Publish-Subscribe Systems
 - Complex Event Processing Systems
- Stream Processing Systems

Distributed Computing Systems

- **Distributed systems**
 - Multiple computers interacting through a computer network to support a single application or a family of interconnected applications – i.e., a ***distributed application***
- **Why Distributed Systems?**
 - Applications require coordinated use of several computers
 - Data generated in one location and needed in another location
 - **Cost:** more cost-efficient than single high-end computer
 - **Reliability:** increased reliability without single point of failure
 - **Scalability/Management:** easier to expand and manage

Distributed Systems – A Brief History

- Concurrent processes
 - Inter-process communication by ***message-passing*** in operating system architectures studied in 1960s
- ARPANET: late 1960s
- Local-area networks
 - Ethernet in 1970s
- ARPANET e-mail early 1970s
- Usenet and FidoNet from 1980s
 - distributed discussion systems

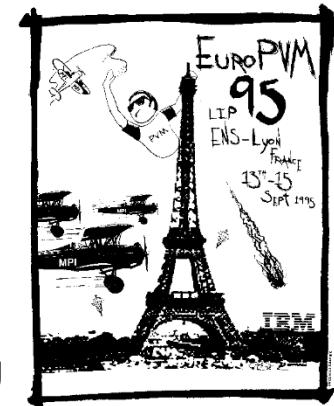


Distributed Systems



- Multiple issues (versus sequential programs)
 - Algorithmic issues
 - Non-trivial design of parallel algorithms
 - “Mechanical” issues
 - Running, debugging, optimizing applications
- Multiple paradigms developed
 - Algorithmic issues
 - Still on our own, in most cases... 😊
 - Certain classes of problems (in computational physics, finance, visualization) have well-defined and friendly programming environments with pre-programmed efficient algorithms underneath them
 - “Mechanical” issues
 - Addressed by different programming environments
 - Language and management constructs
 - Two popular infrastructures: PVM and MPI

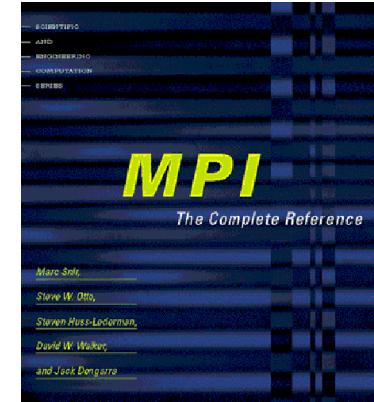
Distributed Systems – PVM



- Parallel Virtual Machine (PVM): software for parallel networking of loosely coupled computers
 - Network of machines can be used as a single distributed parallel processor:
 - Run-time environment and library for message-passing, Task and resource management, Fault notification
 - Functions for **manually** parallelizing an existing source program, or for writing new parallel/distributed programs.
- Process-based computation: Unit of parallelism is a **task**
 - Independent sequential thread of control with communication and computation
 - Multiple tasks may execute on a single processor
- Explicit message-passing model
 - Collections of computational tasks
 - Parallelization strategy: data-, task-, or hybrid decomposition
 - Tasks cooperate by explicitly sending and receiving messages
- Heterogeneity support
 - supports heterogeneity in terms of machines, networks, and applications
- Multiprocessor support
 - Native message-passing facilities on multiprocessors
 - Vendors often supply their own optimized PVM for their systems

```
#include "pvm3.h"
main() {
    int ptid, msgtag;
    char buf[100];
    ptid = pvm_parent();
    strcpy(buf, "hello, world from ");
    gethostname(buf + strlen(buf), 64);
    msgtag = 1;
    pvm_initsend(PvmDataDefault);
    pvm_pkstr(buf);
    pvm_send(ptid, msgtag);
    pvm_exit();
}
```

Distributed Systems – MPI



- Message Passing Interface (MPI) ***specification***
 - Allows many computers and processors to work jointly for an application
 - Used in computer clusters and supercomputers.
 - dominant model for high-performance computing
 - Supports heterogeneous architectures
- Interface provides
 - Virtual topology management
 - Synchronization facilities
 - Communication functionality
- Library functions
 - Point-to-point rendezvous-type send/receive operations
 - Broadcast messages
 - Choice of Cartesian or graph-like logical process topology
 - Data (send/receive) between process pairs
 - Combining partial results of computations (gathering and reduction operations)
 - Synchronizing nodes (barrier operation)
 - Obtaining network-related information

```
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);

if(myid == 0) {
    printf("%d: We have %d processors\n", myid, numprocs);
    for(i=1;i<numprocs;i++)
    {
        sprintf(buff, "Hello %d! ", i);
        MPI_Send(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD);
    }
    for(i=1;i<numprocs;i++)
    {
        MPI_Recv(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD, &stat);
        printf("%d: %s\n", myid, buff);
    }
}
else {
    /* receive from rank 0: */
    MPI_Recv(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD, &stat);
    sprintf(idstr, "Processor %d ", myid);
    strcat(buff, idstr);
    strcat(buff, "reporting for duty\n");
    /* send to rank 0: */
    MPI_Send(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD);
}
MPI_Finalize();
```

Other Distributed Computing Trends

- **Hardware**
 - Multi-core
 - A single chip, multiple “independent” cores
 - Distributed/Hybrid infrastructure
 - Clusters, virtual machines, hybrid configurations, accelerators (GPUs, FPGAs, TPUs)
 - Inter-operation
 - Complex architectures of inter-connected hardware platforms
- **Software**
 - Large-scale inter-operation issues
 - Web-based, middleware-based, databases, solvers, etc...
 - CORBA, DCOM, RPC
 - Integration
 - Business intelligence frameworks
 - Web frontend, databases, component-based integration
 - Different parts written using different languages

Data Management Systems: Databases

Traditional Computing

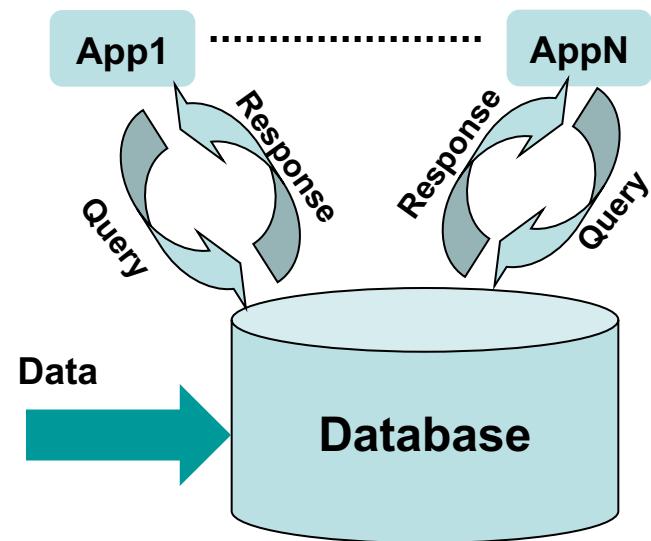


Historical fact finding with data-at-rest

Batch paradigm, pull model

Query-driven: submits queries to static data

Relies on Databases, Data Warehouses



Traditional Computing with Databases

- Databases
 - Software based containers to collect and store information
 - Support retrieval, updates, addition, and removal of data
 - Multiple models: Network, Hierarchical, Relational
- Relational Databases
 - Use a relational model (schema)
 - Provides declarative method to specify data and queries (predicates)
 - Database management takes care of underlying data structures and retrieval mechanisms
 - Most popular types of databases
 - e.g. IBM DB2, Oracle Database, Microsoft SQL Server, PostgreSQL, MySQL

Traditional Computing with Databases

Diagram illustrating Table A (Relation):

The diagram shows a table with a header row and three data rows. The header row contains four columns labeled ID, Name, Age, and Address. The data rows are highlighted in teal. A blue arrow labeled "Header" points to the top row. A green arrow labeled "Tuple/Row" points to one of the data rows. A white arrow labeled "Attribute/Column" points to the column "Name".

ID	Name	Age	Address

Support creation of indices to make data access more efficient

Diagram illustrating Table B (Relation):

The diagram shows a table with three columns labeled ID, Salary, and Taxes. The first column, ID, is highlighted in teal and has a dashed border around it. A white arrow points to this column with the label "Key column may be used to link data from multiple tables".

ID	Salary	Taxes

Key column may be used to link data from multiple tables

- Queries against database specified in Structured Query Language (SQL)
- Operations
 - Union – combine tuples of two relations (remove duplicates)
 - Intersection – find common tuples
 - Difference – set difference
 - Cartesian product
 - Select – subset of tuples from relation
 - Project – Select with duplicates removed (group by, distinct)
 - Join – tuples from two relations are merged based on common attribute (key)
 - Division – inverse of cartesian product (use tuples from one relation to partition tuples in another relation)
- Stored Procedures: Executable code stored in the database to customize operations

Traditional Computing with Databases

- Relational databases used widely
 - Financial records, Manufacturing, Personnel
- Several analytic tools developed
 - Online Transaction Processing (OLTP)
 - Manage transaction oriented applications
 - Online Analytical Processing (OLAP)
 - Extensions with multi-dimensional databases and efficient data aggregates to answer complex queries efficiently
 - Data Mining Tools
 - SAS, SPSS, Weka
 - Visualization and Report Generation Tools
 - Cognos

NoSQL Stores: Semi-Structured Data

- Provide data storage outside tabular form
- Scale better than SQL Databases
 - Sacrifice Atomicity Consistency Isolation Durability (ACID) guarantees for eventual-consistency
- Can store semi-structured data
- Often provide SQL-like languages for query
- Several Types
 - Column: Accumulo, Hbase, Cassandra
 - Document: CouchDB, Cosmos DB, MongoDB,
 - Key Value: Redis, BerkeleyDB, Zookeeper
 - Graph: Apache Graph, Neo4J

Databases and Streaming Data (?)

- Data Ingest
 - Cannot easily handle unstructured data types and proprietary formats, including audio, video, multimedia, graph structures etc.
 - Notion of time and data order not natural
- Data Analysis
 - Mining tools available, but hard to extend and customize
 - Limited support for long-running queries
- Performance
 - Cannot keep up with data rates, analysis requirements, latency
 - Cannot scale to store the potentially infinite amount of stream data
- Ease of Use
 - Requires using SQL or C++/Java extensions to invoke SQL
- Solutions based on traditional computing face severe bottlenecks
 - Applications often run hours to days behind data
 - Majority of collected data is never analyzed

From Databases towards Streaming

- Active databases (e.g. triggers in commercial Databases)
- ECA (event-condition-action) rules
 - capture events, the conditions surrounding these events, and the actions to be triggered when the conditions are met
 - Implemented as SQL triggers
- Closed
 - Only operate on events within database
 - Ode, HiPac
- Open
 - External event sources are allowed
 - Samos, Snoop
- Disadvantages
 - Data always needs to be stored before processing

Continuous Query Systems

- Continuous queries
 - Standing queries that run until explicit termination
 - Trigger, Query, Stop Condition
 - “monitor the price of 10 megapixel (MP) digital cameras in next two months and notify me when one with a price less than \$100 becomes available”
- Evaluated continuously over changing data
 - XML based file input
- Push-based model (answers sent to user)
 - Rule based analytics
- NiagaraCQ, OpenCQ
- Precursors to Stream Processing
 - Run data through queries ☺

Publish/Subscribe Systems

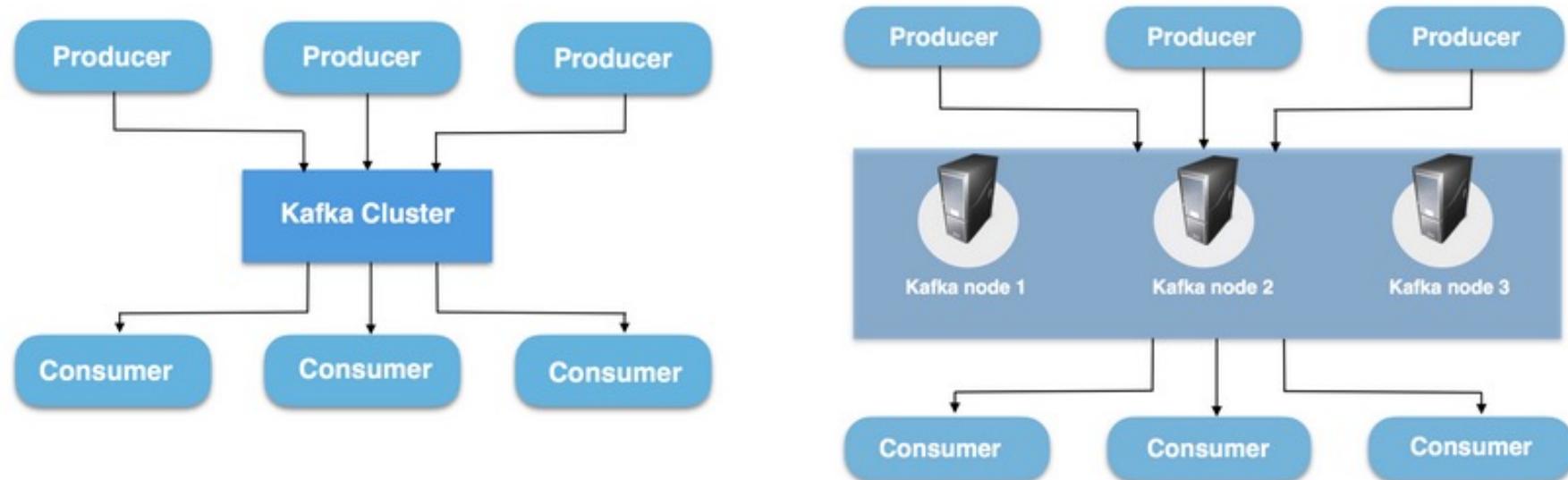
- Asynchronous messaging systems
 - Senders publish messages (characterized into classes) without sending to specific receivers
 - Receivers subscribe to different classes to receive messages of interest
 - Intermediate message broker forwards messages
- Decoupling of publisher and subscriber
 - Improved scalability
 - Dynamic network topology
- Message oriented middleware solutions
 - Message queue, Java Message Service
- Message filtering
 - Topic based: Logical channels, all messages in topic received
 - Content based: Message filtered based on attribute
 - Hybrid possible

Publish/Subscribe Systems

- **Systems**
 - Gryphon, Siena - U. Colorado, Padres - U. Toronto
- **Extensions into Enterprise Service Bus**
 - SOAP, Web-services etc.
- **Disadvantages**
 - Cannot guarantee delivery of data
 - Instabilities in throughput
 - Bursts of data followed by long periods of silence
 - Scalability issues
 - Slowdowns as number of applications increase
 - Lack of security

Publish/Subscribe Systems

- Apache Kafka: distributed commit log service that functions much like a publish/subscribe messaging system
 - Improved throughput,
 - built-in partitioning,
 - replication and fault tolerance



<https://www.cloudkarafka.com/blog/2016-11-30-part1-kafka-for-beginners-what-is-apache-kafka.html>

Publish/Subscribe Systems: Kafka

Kafka Topic

A **Topic** is a category/feed name to which messages are stored and published. Messages are byte arrays that can store any object in any format. As said before, all Kafka messages are organized into topics. If you wish to send a message you send it to a specific topic and if you wish to read a message you read it from a specific topic. Producer applications write data to topics and consumer applications read from topics. Messages published to the cluster will stay in the cluster until a configurable retention period has passed by. Kafka retains all messages for a set amount of time, and therefore, consumers are responsible to track their location.



<https://www.cloudkarafka.com/blog/2016-11-30-part1-kafka-for-beginners-what-is-apache-kafka.html>

Publish/Subscribe Systems: Kafka

Kafka topic partition

Kafka topics are divided into a number of partitions, which contains messages in an unchangeable sequence. Each message in a partition is assigned and identified by its unique **offset**. A topic can also have multiple partition logs like the *click-topic* has in the image to the right. This allows for multiple consumers to read from a topic in parallel.



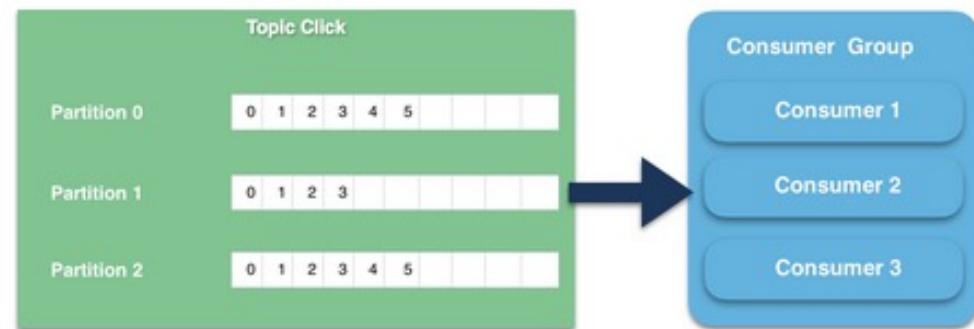
In Kafka, replication is implemented at the partition level. The redundant unit of a topic partition is called a replica. Each partition usually has one or more replicas meaning that partitions contain messages that are replicated over a few Kafka brokers in the cluster. As we can see in the pictures - the *click-topic* is replicated to Kafka node 2 and Kafka node 3.

<https://www.cloudkarafka.com/blog/2016-11-30-part1-kafka-for-beginners-what-is-apache-kafka.html>

Publish/Subscribe Systems: Kafka

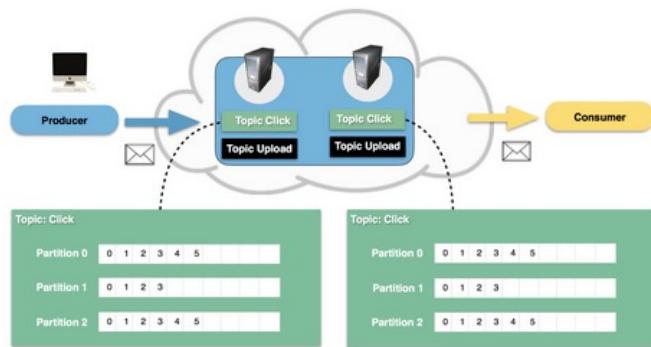
Consumers and consumer groups

Consumers can read messages starting from a specific offset and are allowed to read from any offset point they choose. This allows consumers to join the cluster at any point in time.



<https://www.cloudkarafka.com/blog/2016-11-30-part1-kafka-for-beginners-what-is-apache-kafka.html>

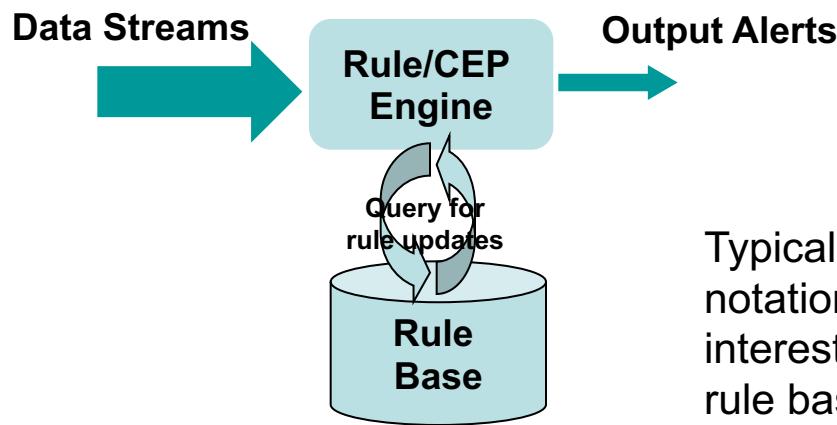
Publish/Subscribe Systems: Kafka



1. *A user with user-id 0 clicks on a button on the website.*
2. *The web application publishes a message to partition 0 in topic "click".*
3. *The message is appended to its commit log and the message offset is incremented.*
4. *The consumer can pull messages from the click-topic and show monitoring usage in real-time, or it can replay previously consumed messages by setting the offset to an earlier one.*

<https://www.cloudkarafka.com/blog/2016-11-30-part1-kafka-for-beginners-what-is-apache-kafka.html>

Complex Event Processing Engines



Typically accept condition/action pairs using “if-then” notation and watch input stream for condition of interest. Enforce collection of rules/patterns stored in rule base.

Rule firing leads to alerts, as well as other rules becoming active

CEP Systems: Rules/Patterns

- Temporal sequences
 - event *A* following event *B*
- Negation
 - event *A* not appearing in a sequence
- Kleene closure
 - event *A* appearing repeatedly a number of times
 - Regular expression based search
- Support for time windows

Example Rules

Intrusion Detection Rules

GeneralBruteForceTwentyInTwoUniqDST: Alert anytime that a single *destination IP* targeted with **20** or more events in “generalbruteforce” group *within a 2 minute window*

HttpScanFiveInThreeUniqSRCBootstrapRule: Alert anytime that any single *source ip* produces more than **5 unique event names** in the “httpscan” group *within a 3 minute window*

NotificationNeededForP2PRule: Alert each time *an event in “p2p” group* is detected

- Expert defined rules on data streams
- Simple rules
 - Comprehensibility and Ease of Construction
 - User Validation
- Typical Analytics
 - Capture event/group arrival rates, temporal relationships, filter and composite conditions

Telco Mediation Rules

```
RULE TR33_A{
  (CALL_RECORD_TYPE==1) || (CALL_RECORD_TYPE==3) ||
  (CALL_RECORD_TYPE==5) || (CALL_RECORD_TYPE == 15)
=>
  CALL_DIRECTION = "O";
}
RULE TR33_B{
  (CALL_RECORD_TYPE==2) || (CALL_RECORD_TYPE==4) ||
  (CALL_RECORD_TYPE==7) || (CALL_RECORD_TYPE == 14)
=>
  CALL_DIRECTION = "I";
}
RULE TR33_C{
  (CALL_RECORD_TYPE == 0)
=>
  CALL_DIRECTION = "T";
}
RULE TR33_D{
  VAR X = lookup(incomingRoute, "DIM_MCS_ROUTE","ROUTE")
  (CALL_RECORD_TYPE == 0) && (incomingRoute != NULL) && (X)
=>
  CALL_DIRECTION = "R";
}
RULE TR33_E{
  VAR X = lookup(incomingRoute,"DIM_MCS_ROUTE","ROUTE")
  VAR Y = lookup(outgoingRoute,"DIM_MCS_ROUTE","ROUTE")
  (CALL_RECORD_TYPE == 0) && (incomingRoute != NULL) &&
  (outgoingRoute != NULL) && X && Y
=>
  CALL_DIRECTION = "T"; }
```

CEP Engines for Streaming Data

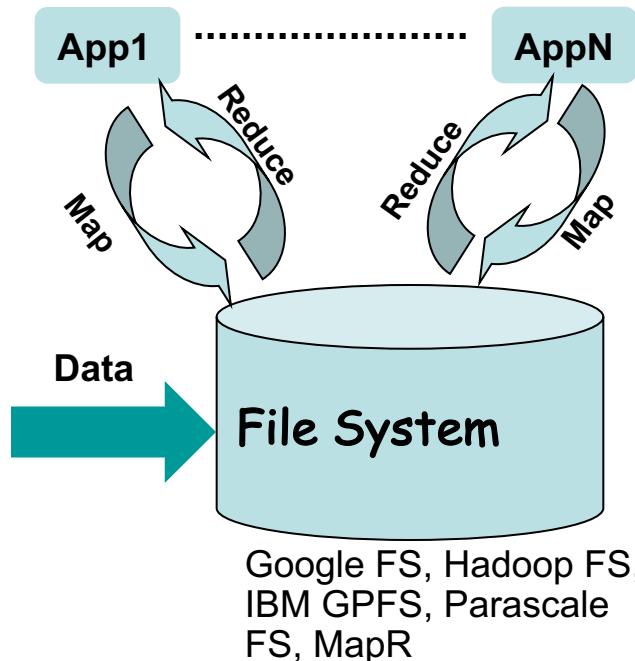
- Several popular engines
 - IBM ODM, Oracle CEP, Tibco Business Events
- Data Ingest
 - Cannot easily handle unstructured data types and proprietary formats, including audio, video, multimedia, graph structures etc.
 - Cannot easily handle data from heterogeneous and distributed sources
- Data Analysis
 - Limited to simple rule languages, need to be extended to support stream processing requirements
- Performance
 - Cannot handle very large data rates and low latency
- Ease of use
 - Low barrier to usage – several UIs available

Other Systems

- Supervisory Control And Data Acquisition (SCADA) systems
- Extract/Transform/Load (ETL) systems

Map-Reduce Framework

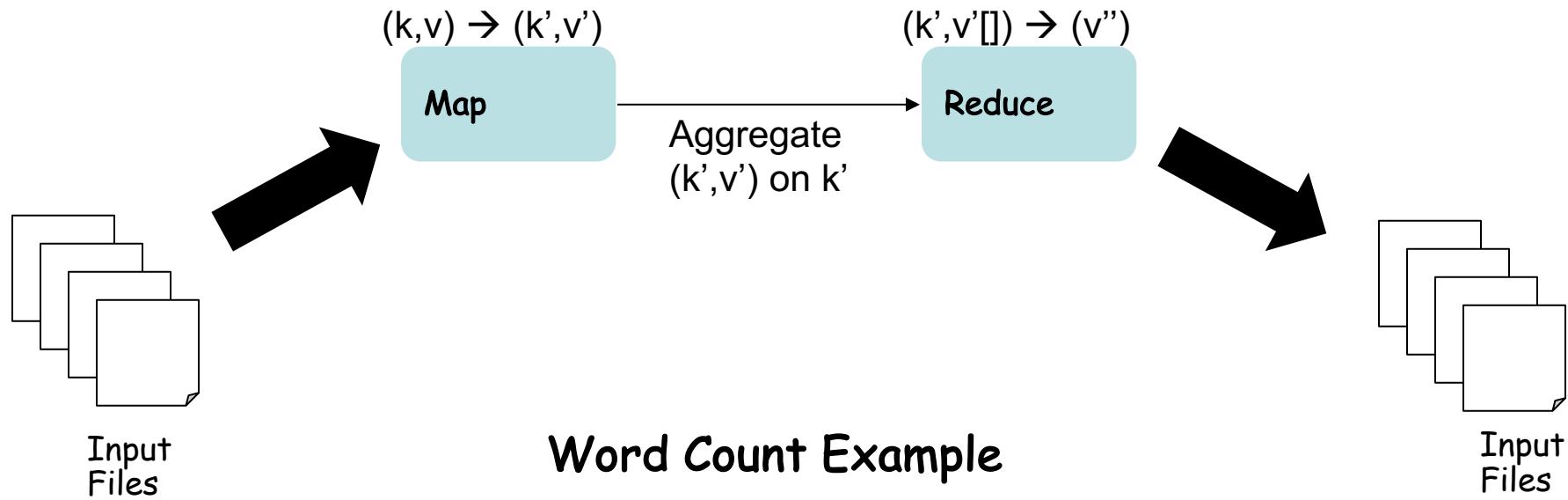
Applications partitioned into Map/Reduce and implemented on distributed compute infrastructure



- Programming model (from Google) for large-scale distributed 'batch' processing
 - Powerful paradigm but restricted to certain classes of problems
 - Extensible for different applications
- Execution environment
 - Exploits parallelism (pipelined and data)
 - Resilience to failures and jitter

Inspired by Map/Reduce paradigm in functional programming (e.g. Lisp).

Map-Reduce Programming Model

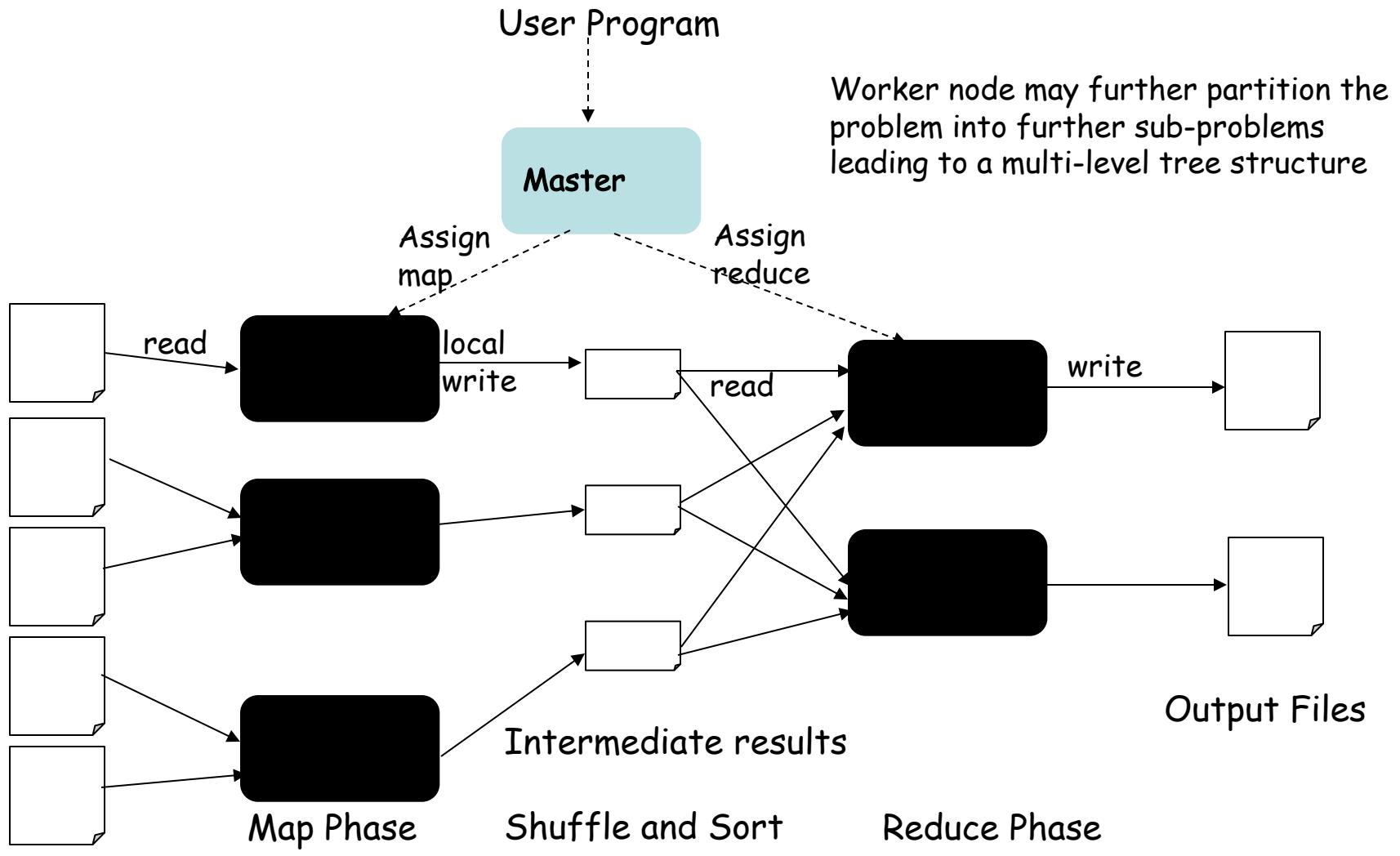


Word Count Example

```
void map(String name, String document):
    // name: document name
    // document: document contents
    for each word w in document:
        EmitIntermediate(w, "1");
```

```
void reduce(String word, Iterator<String> partialCounts):
    // word: a word
    // partialCounts: a list of
    aggregated partial counts
    int sum = 0;
    for each pc in partialCounts:
        sum += ParseInt(pc);
    Emit(word, AsString(sum));
```

Map-Reduce Execution Model



Using Map-Reduce Programming Model

- Input Reader
 - Divide input into appropriate size splits and generate key/value pairs
- Mapper
- Partition
 - Allocate each mapper result to appropriate reducer
- Compare
 - Used to sort/shuffle the intermediate results for improved efficiency
- Reducer
- Output Writer
 - Write results to output file system
- Google's use of Map-Reduce*
 - Clustering of news items for Google News, Processing of satellite imagery, Statistical machine translation, Large-scale machine learning

*J. Zhao, J. Pjesivac-Grbovic, "MapReduce The Programming Model and Practice", SigMetrics 2009.

Hadoop

- Apache Implementation of the Map-Reduce Framework
 - Open source Java implementation
 - Yahoo! significant contributor
- Hadoop File System
 - Distributed, Scalable, Portable Java based filesystem
- Job Tracker
 - Partitions work out to individual task trackers
 - Monitors task trackers via heartbeat
- Task Tracker
 - Spawns processes for the job
- Scheduler
 - FIFO, Fair (Facebook), Capacity (Yahoo!)
- Fault Tolerance
 - Limited fault tolerance

Machine Learning Map-Reduce: Mahout

- Mahout Open Source Learning Library
 - Apache
- Supports development of scalable machine learning libraries on Hadoop
- User-contributed (somewhat large community)
- Two modes of operation
 - single-mode and map-reduce
- Command-line based analogous to Hadoop

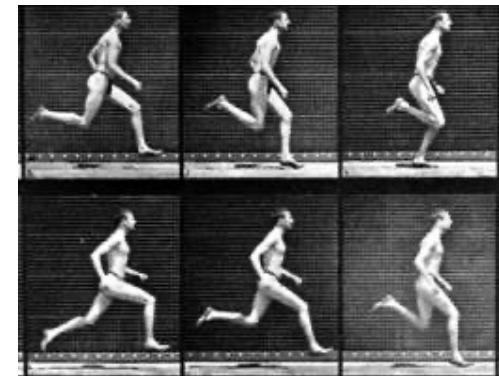
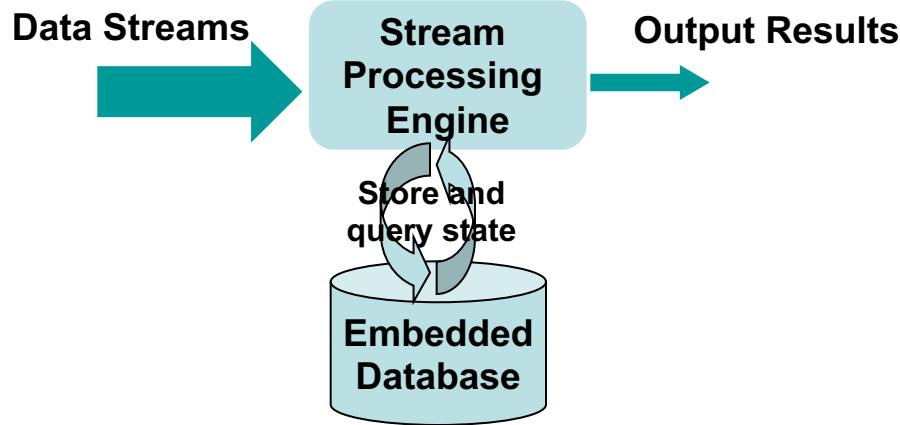
```
$MAHOUT_HOME/bin/mahout kmeans -I <in> -o <out> -k 50
```

- API:

```
KMeansDriver.runjob(Pathinput, Pathoutput...)
```

- Supported Algorithms
 - Classification (Logistic Regression, Naïve Bayes, C-Bayes, Random Forest)
 - Clustering (Kmeans, fuzzy kmeans, spectral clustering, Canopy, Fuzzy K-means, dirichlet)
 - Frequent Pattern Mining (Pattern and itemset mining)
 - Dimensionality Reduction (SVD)
 - High performance collection (opted from CERN Colt library)
 - Math library

Stream Processing Systems

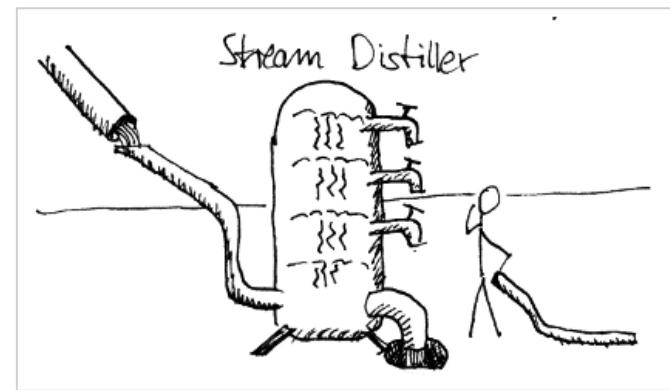
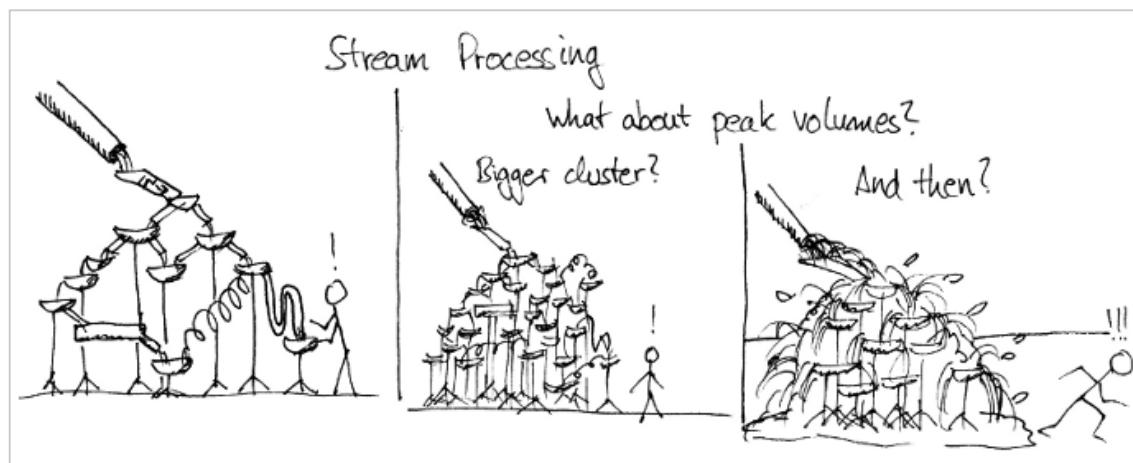
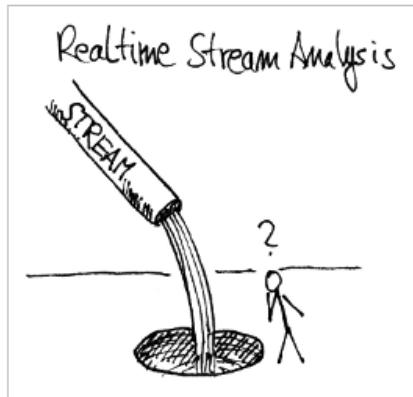


Real time analysis of data-in-motion

Perform arbitrarily complex processing on data as it streams through. Use embedded database to store data summaries, history, state information etc.

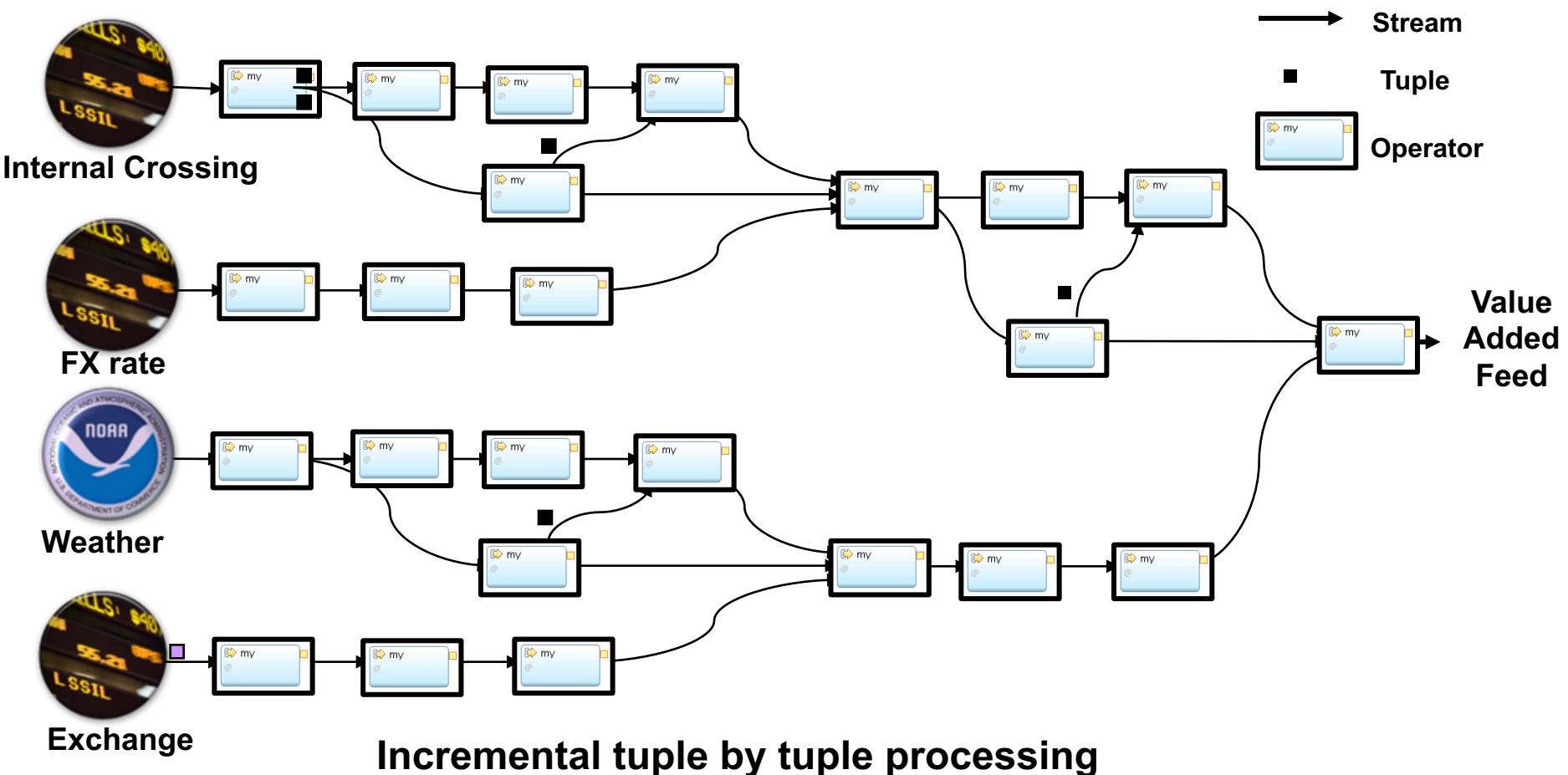
Unlike DBMS long-running queries through which data flows to get filtered appropriately

Databases versus Stream Processing

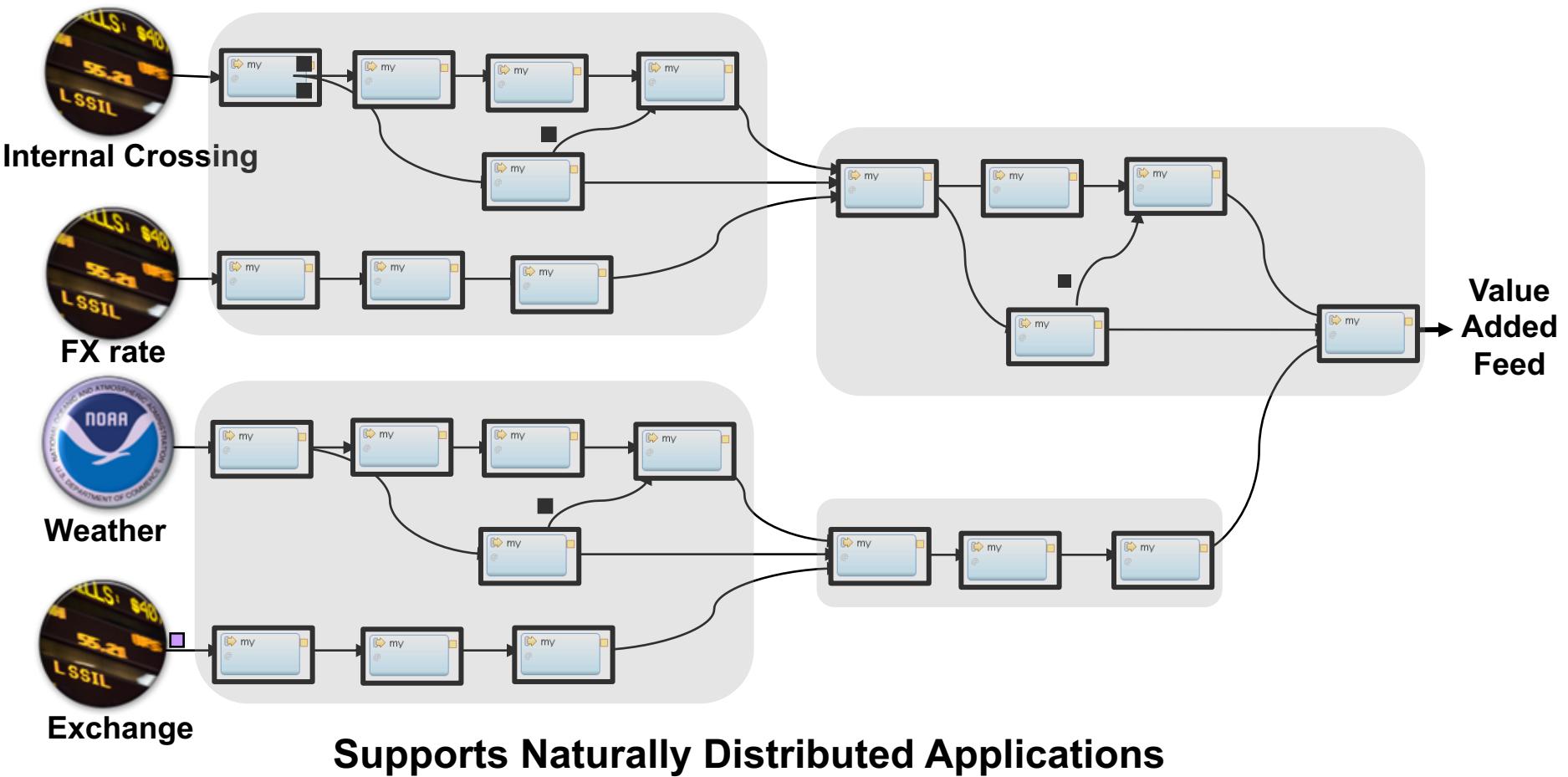


Thanks to B. Gedik

Introduction to Stream Processing



Introduction to Stream Processing



Stream Processing System Principles

- Keep Data Moving
 - Process message in stream without requiring storage
 - Use asynchronous, non-polling based communication
- Handle Stream Imperfections
 - Out of order, missing, delayed, noisy data
- Generate predictable outcomes
 - Reproducible results
- Handle both streaming as well as stored data
 - Efficient storage, access and modification of state information
- Support data safety and availability
 - High availability and resilience to loss – when needed
- Partition and scale applications
 - Distribute processing to achieve incremental scaling
- Process and respond instantaneously
 - Optimized and high performance execution engine
- Query using high level languages
 - Do not rely purely on C++/Java to minimize development time

Stream Processing Systems

- Programming Model
 - Flowgraph composition
 - Development Environment
- Runtime
 - Distribution
 - Data transport
 - Scheduling
 - Fault Tolerance
- Toolkits
 - Connectors
 - Analytics
 - Stream Relational

Stream Processing Systems

- Academic Systems (now retired)
 - TelegraphCQ – Berkeley
 - Aurora & Borealis – Brown
 - STREAM – Stanford
- Industrial Prototypes
 - Gigascope – AT&T
 - System S – IBM Research
- Open Source Systems
 - S4 – Yahoo S4!
 - Storm – Twitter, Apache
 - Apache Flink
 - **Apache Spark Streaming**
 - **Apache Beam**
- Commercial Systems
 - StreamBase – StreamBase Systems
 - Amazon Kinesis
 - IBM Streams
 - Google Dataflow

Stream and Signal Processing

- Sensing and Aggregation of Data from Diverse Sources
 - Task Driven Sensing
 - Robust and Error Resilient Data Gathering
 - Distributed compression, data reduction, processing
- Managing confidence, uncertainty and noise
 - Missing samples, delayed samples, non-time aligned
 - Algorithms and system services to support analytics/application
- Resource Constrained Online Analytics
 - Resource adaptive filtering, tracking, feature extraction, compression
 - Complexity scalable mining and online learning
 - Distributed analysis

Going Forward

- Examples of two Stream Processing Systems
 - Apache Spark Streaming, Apache Beam
- Fundamentals of stream processing systems
 - Plumbing and architecture of systems
 - Programming stream processing applications
 - Design principles of application development
- Streaming Data Analysis
 - Data Preprocessing, Data Analysis
- Focus on exploring space of stream processing
 - Student seminars and projects

Anatomy of A Stream Processing System

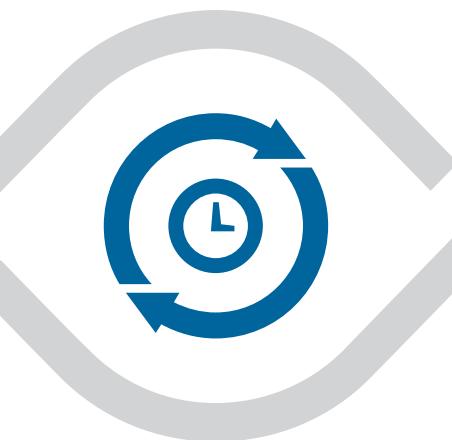
Integrated Development Environment

Scale-Out Runtime

Analytic Toolkits



Development and Management

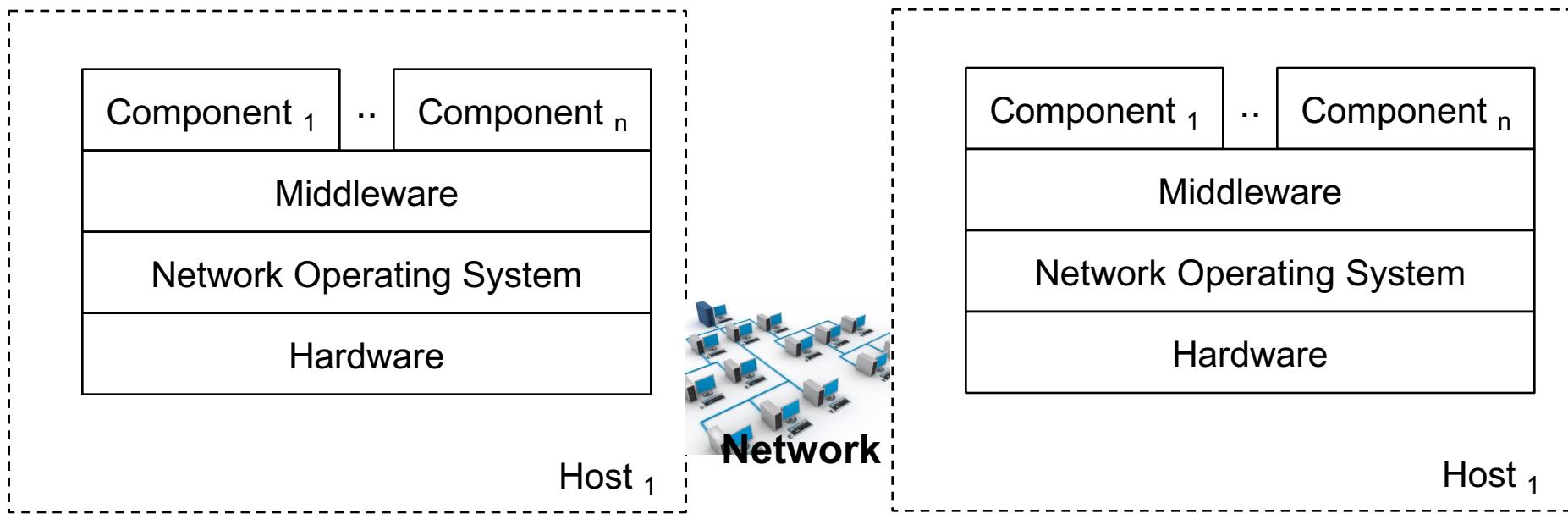


Flexibility and Scalability



Functional and Optimized

Middleware



- *Middleware*: computer software that provides services to software applications beyond those available from the operating system
- *Stream Processing Middleware*: distributed system that provides an execution substrate for stream processing applications

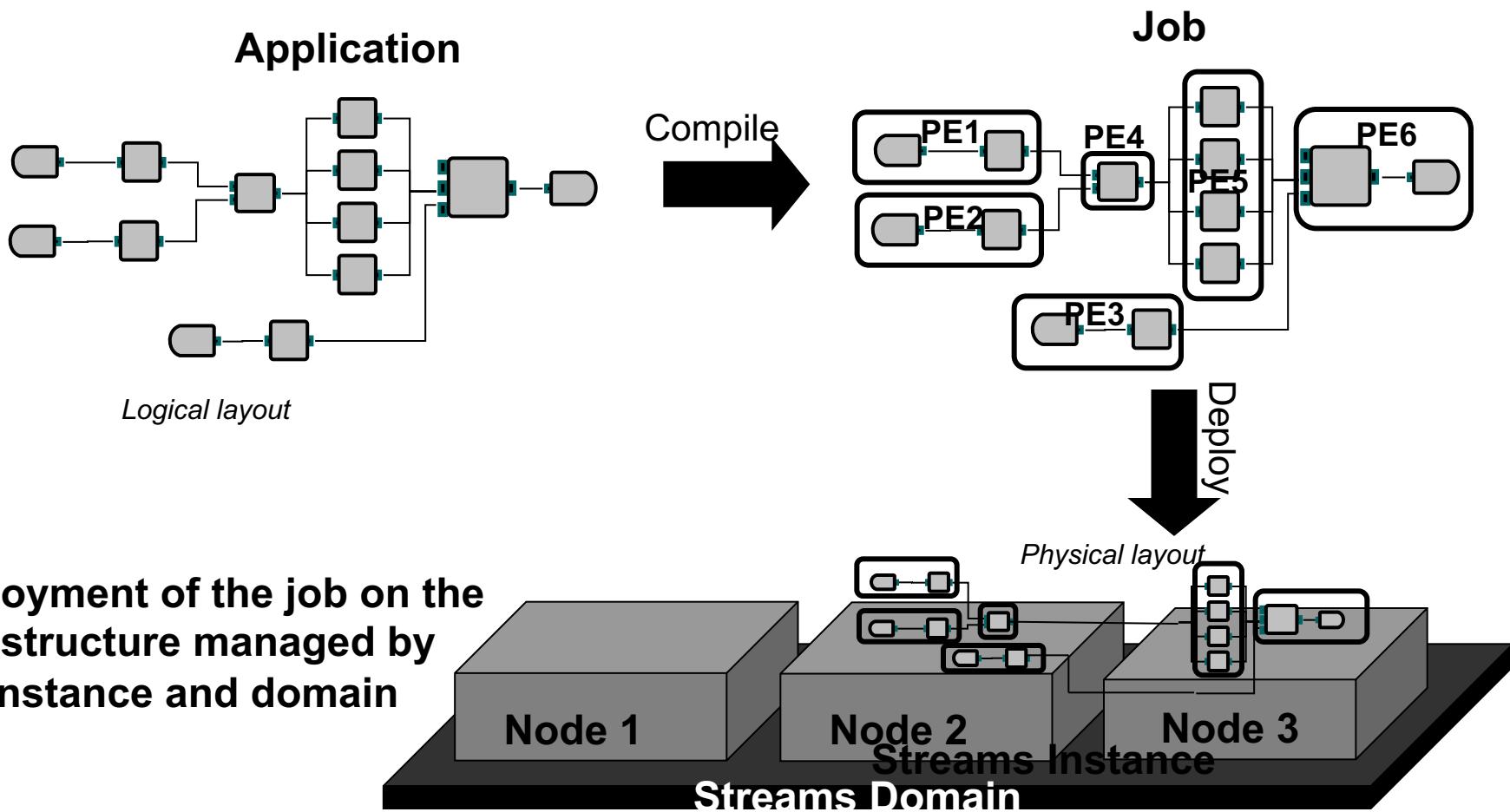
Middleware Support

- Access transparency
 - Local and remote components accessible via same operations
- Location transparency
 - Components locatable via name, independent of location
- Concurrency transparency
 - Components/objects can be run in parallel
- Migration transparency
 - Allows movement of components without affecting other components
- Replication transparency
 - Allows multiple instances of objects for improved reliability (mirrored web pages)
- Failure transparency
 - Components designed while accounting for failure of other services

Conceptual Entities of Interest

- *Domain*
 - Grouping of resources for common management
- *Instance of a stream processing middleware*
 - runs a number of applications/jobs within domain
- *User*
 - Application developer, application analyst, or system administrator
- *Application*
 - A logical data flow graph
- *Job*
 - Instantiated version of the application
- *Processing Element (PE)*
 - Corresponds to an OS process
 - Execution container for a sub-graph

Conceptual Entities of Interest



Stream Processing Middleware Services

- Distribution
 - Plumbing related to moving processes around
- Resource management
 - Placement, scheduling, load shedding
- Job management
 - Starting/stopping/editing jobs, handling dynamic connections
- System monitoring
 - Health, state, performance of the system
- Security
- Logging
- Data Transport
 - Inter-PE, intra-PE, intra-host, inter-host
- Debugging
- Visualization

Instance Components

Component Name	Acronym	Executable Name
Streams Application Manager	SAM	<code>streams-sam</code>
Streams Resource Manager	SRM	<code>streams-srm</code>
Scheduler	SCH	<code>streams-sch</code>
Name Service	Ns	<code>dnameserver</code>
Authentication and Authorization Service	AAS	<code>streams-aas</code>
Streams Web Server	Sws	<code>streams-sws</code>

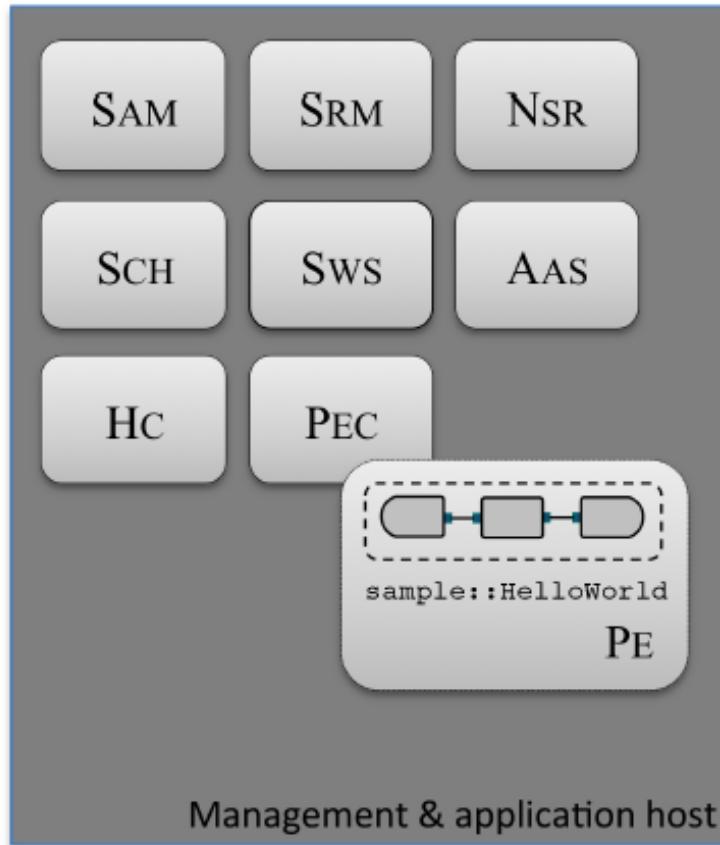
Domain Services

Table 8.1 InfoSphere Streams centralized components.

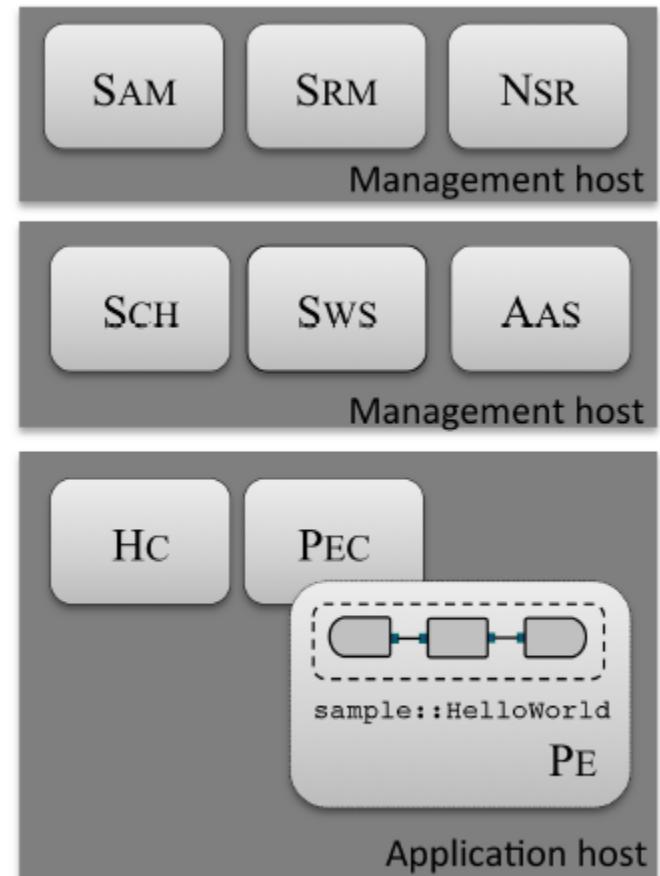
Component Name	Acronym	Executable Name
Host Controller	Hc	<code>streams-hc</code>
Processing Element Container	PEC	<code>streams-pec</code>

Table 8.2 InfoSphere Streams distributed components.

Middleware Components



Single Node Instance



Multiple Node Instance

Application and Job Management

- A user submits an application to the runtime instance
- Runtime checks security credentials to ensure a job can be instantiated by the user for this instance
- A job with an id is created from the application
- The scheduler is tasked with placing the PEs (typically formed at compile-time) on the hosts of the instance
 - Considering current load of the system
 - Considering the placement constraints
- The PEs are then moved to the hosts and instantiated there
- The connections between the PEs are established via the transport
- Additional workflows:
 - Job cancellation
 - Dynamic connection management
 - PE relocations, restarts

Resource Management

- *Resource manager* needs to track
 - Resource health
 - Is the host alive?
 - Resource amounts
 - How much CPU is available on the host?
- Health is often tracked using liveness pings
 - Check if a resources is alive by sending a periodic ping
 - If not results for a long time, assume its dead
- Resource amounts are tracked using metrics
 - Collect usage metrics periodically to track availability

Application Monitoring

- Application health
 - Is the PE healthy?
 - Yes, no, partially
 - What state is the PE in
 - Initialization, connection establishment, running, etc.
- Application performance
 - Throughput
 - Backpressure (queue sizes or blocking time)
 - Latency (very application specific)
- Custom metrics

Scheduling

- Compile time and Runtime Scheduling
- There could be a few different goals
 - Load balancing
 - Hosts have similar load on them
 - This is more important when the system is highly loaded
 - Application performance
 - Throughput is typically a major concern
- Some difficult issues:
 - Input rates are not known
 - Can predict based on history
 - Seasonality is typical
 - Operator costs/selectivities are not known
 - Profiling runs to measure operator characteristics

Scheduling

- Provide placement
 - Given job/application constraints
 - `partitionColocation`, `partitionExlocation`,
`partitionIsolation`
 - Given current state of the compute infrastructure
- Steps
 - Check constraints – to see if placement feasible
 - Solve bipartite matching problem (`PEs`→hosts)
 - Incrementally learn resource consumption of each PE
 - Load balancing approach: Place in decreasing order of PE complexity on nodes with increasing levels of load
 - i.e. place most expensive PE on least loaded host
 - Periodically re-optimize

Fault Tolerance

- Fault tolerance of the middleware
 - Runtime components save state to database periodically
 - Includes commands, responses with unique IDs
 - Ensures commands executed *at most once* on recovery
 - Service requests use fault tolerant communication with client-server architecture
 - Critical middleware state changes infrequently
 - compared to application state and data
- Application fault-tolerance with data loss
 - Restart failed PEs (with some bound)
 - Relocate PEs on failed hosts
 - Managing state

Application Fault Tolerance

- Active replication
 - Run multiple copies, switch over when you detect failure in one
 - Switch over is a problem for non-deterministic apps
 - Time based windows, arbitrary fan-in
 - Disadvantage: n times resources for n -fold tolerance
 - Given it has little impact on performance, this may be a good choice resource-wise as well, if throughput is the main challenge
- Checkpointing with source data persistence
 - Get a consistent checkpoint, log all source data that is not yet part of a checkpoint
 - Disadvantage
 - costly recovery process
 - operators need to know how to checkpoint their state

Other Services

- Name Service
 - directory for storing references to objects
 - object is location of a component represented as a Common Object Request Broker Architecture (CORBA) Interoperable Object Reference (IOR)
 - Users submit lookup requests for a human-readable symbolic name
 - Can remove and register objects
 - Objects can include data transport endpoints (more later)
- Authentication and Authorization Service
 - provides authorization checks
 - inter-component authentication
 - includes a repository of Access Control Lists (for different entities)
 - Can *entity* perform operation on *object*?
 - E.g. can the user `user` cancel job 2011

Visualization

Commercial Systems: Streams and Dataflow

- Compile time topology visualization
- Run time topology visualization
 - State
 - Health
 - Metrics
 - Logical to physical mapping
- Data visualization
 - Live charts
 - Dashboards

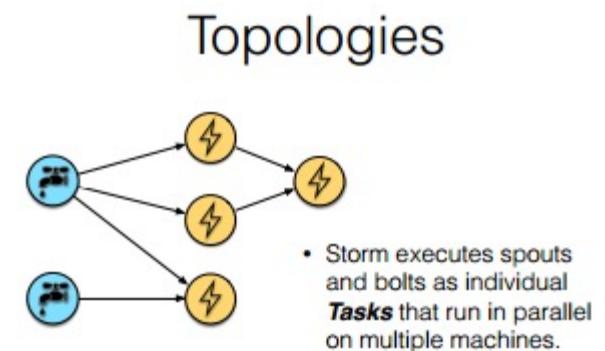
References and Reading

- Chapter 1 of book
- Eight Requirements of Stream Processing:
<http://www.cs.brown.edu/research/db/publications/8rulesSigRec.pdf>
- Aurora: <http://www.cs.brown.edu/research/aurora/>
- Borealis: <http://www.cs.brown.edu/research/borealis/public/>
- Telegraph CQ: <http://telegraph.cs.berkeley.edu/>
- Stanford Stream (inactive): <http://infolab.stanford.edu/stream/>
- Streamit: <http://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TM-620.pdf>
- Streambase: <http://www.streambase.com/>
- Apama: <http://web.progress.com/en/apama/index.html>
- IBM System S/InfoSphere Streams: <http://www-01.ibm.com/software/data/infosphere/streams/>
- Linear Road Benchmark: <http://www.cs.brandeis.edu/~linearroad/>

Backup

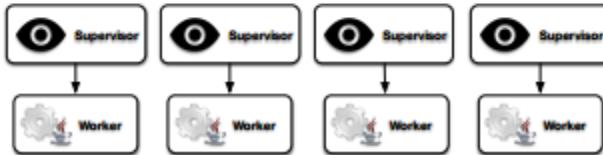
Apache Storm

- A tuple: an immutable set of key / value pairs
- Streams: an unbounded sequence of tuples
- Spouts: source of streams..
- Bolts: receive tuples and do stuff – emit other tuples, read / write data store, perform computation
- Routing tuples between tasks:
 - Shuffle (random)
 - LocalOrshuffle
 - FieldGroupings

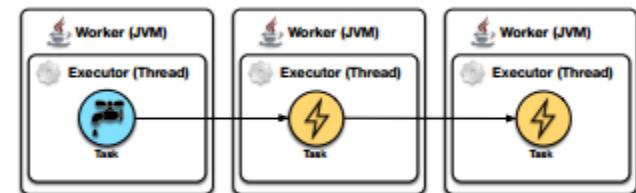
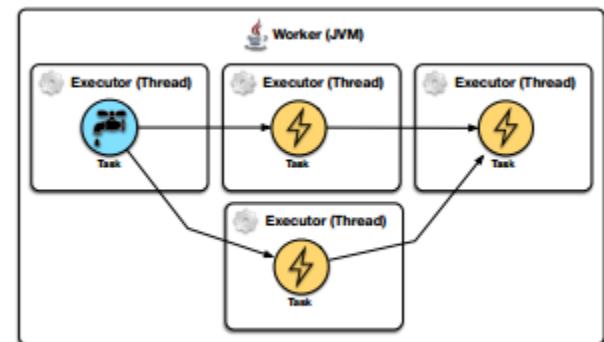
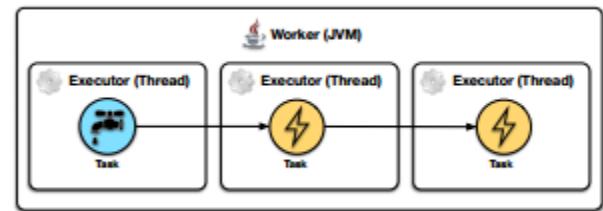


Apache Storm: Topology, Parallelism

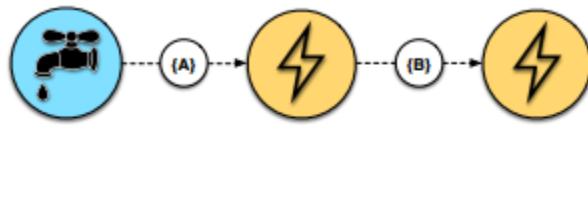
Topology Deployment



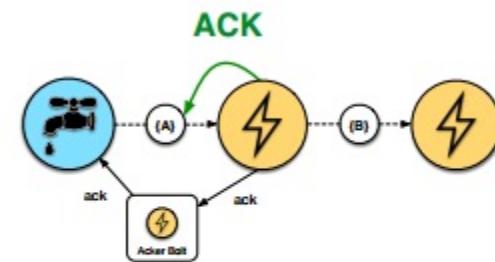
- Built with resilient components
- Parallelism via threads, JVMs



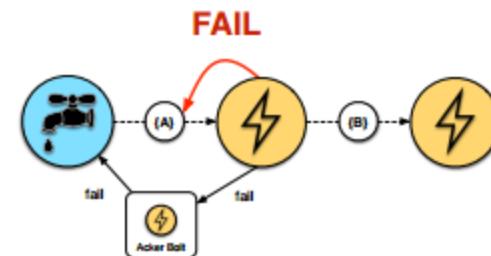
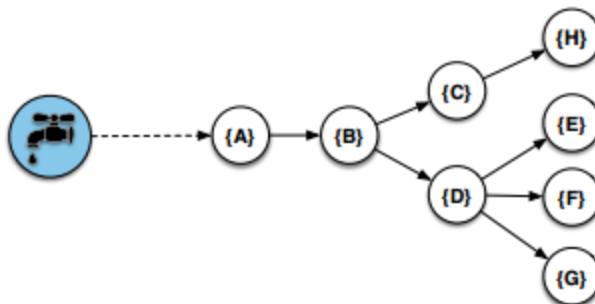
Apache Storm: Reliable Processing



Bolts may emit Tuples **Anchored** to one received.
Tuple "B" is a descendant of Tuple "A"



Acks are delivered via a system-level bolt

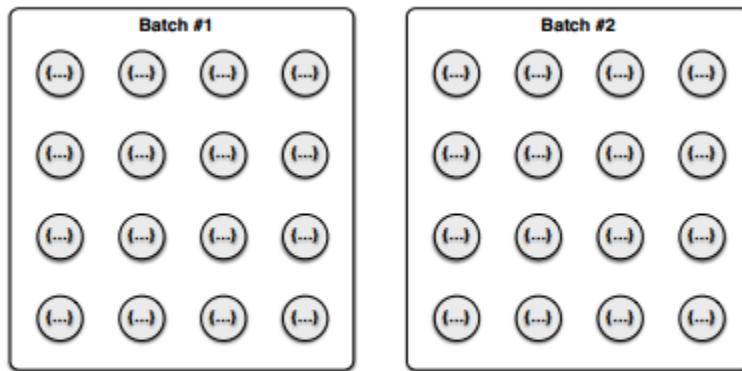


Apache Storm Deployment Models

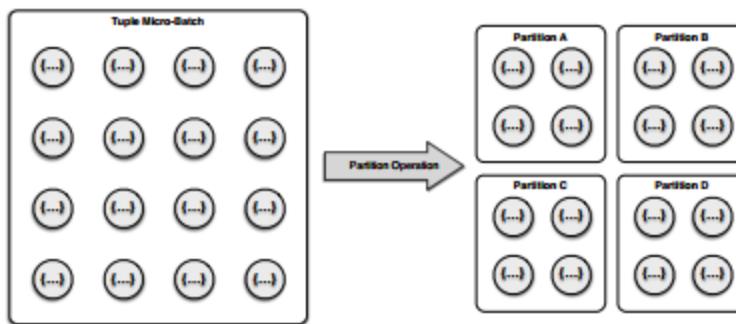
- Core Storm (spouts and Bolts)
 - One a time
 - Low latency
 - Operates on tuple streams
- Trident (streams and operations)
 - Micro-batch
 - Higher throughput
 - Operates on streams of tuple batches and partitions

Apache Storm Trident: Micro-batching

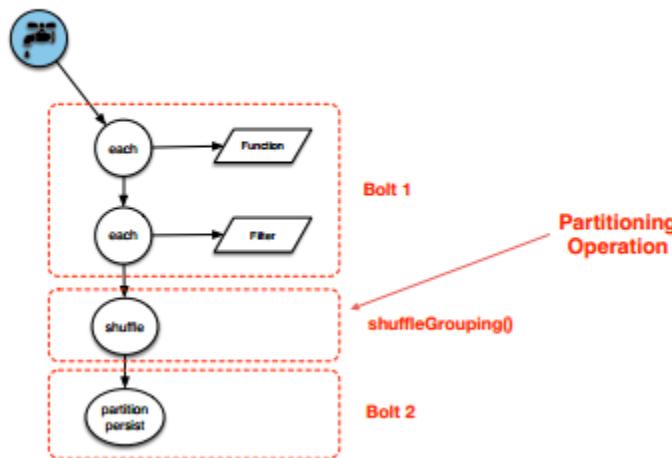
Trident Topologies



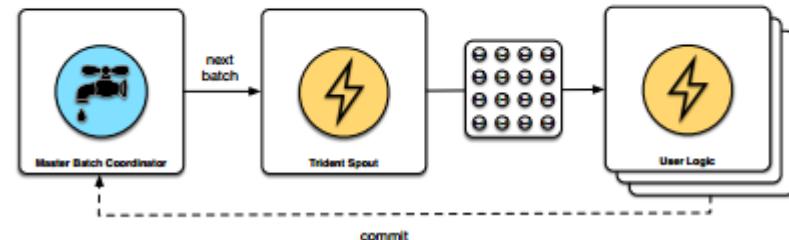
Trident Batches are **Ordered**



Trident Batches can be **Partitioned**



Trident Batch Coordination



Apache Storm

- In use by several companies

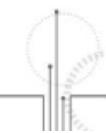


- However, recent development slow
 - Other open source active projects (Spark Streaming)
- More information
 - <https://storm.apache.org/>

Amazon Kinesis: Core concepts

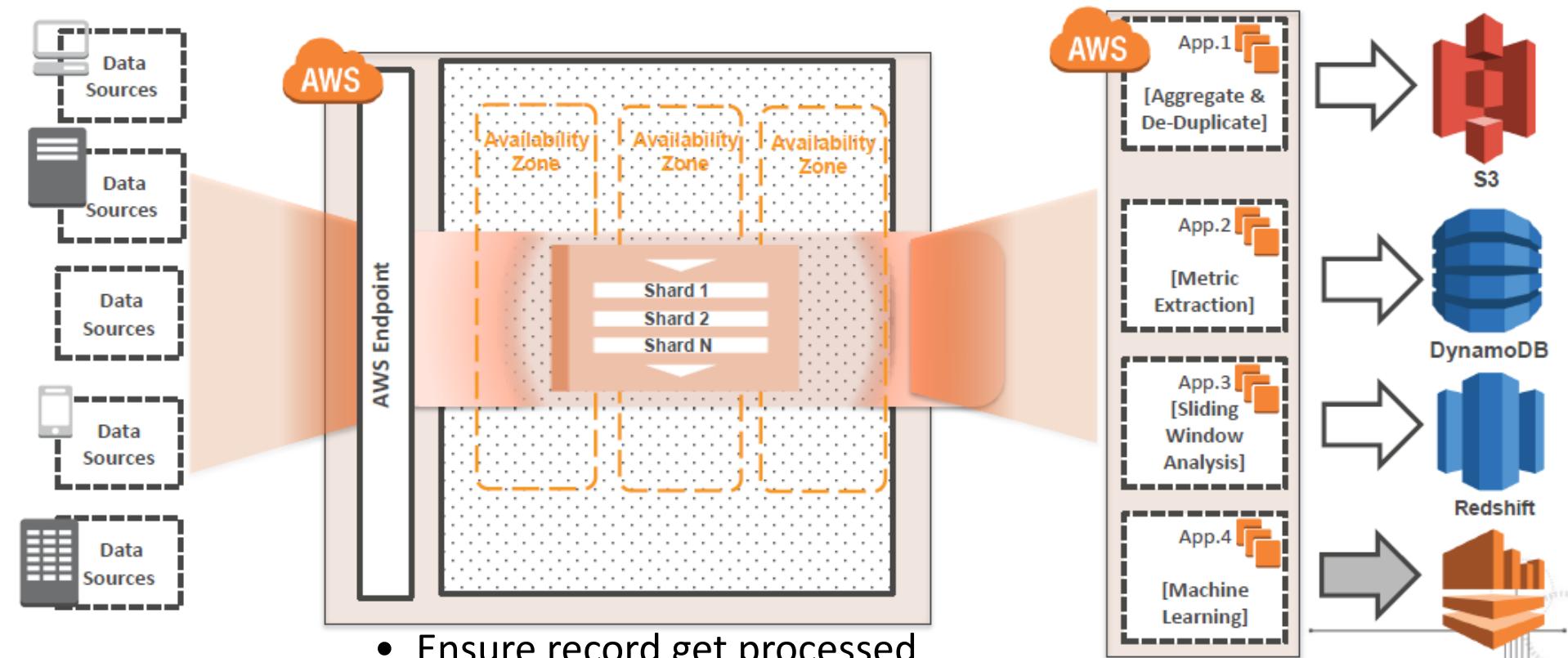
- **Data record** ~ tweet
- **Stream** ~ all tweets (the Twitter Firehose)
- **Partition key** ~ Twitter topic (every tweet record belongs to exactly one of these)
- **Shard** ~ all the data records belonging to a set of Twitter topics that will get grouped together
- **Sequence number** ~ each data record gets one assigned when first ingested.
- **Worker** ~ processes the records of a shard in sequence number order

“A common use is the real-time aggregation of data followed by loading the aggregate data into a data warehouse or map-reduce cluster”



Amazon Kinesis

Service for Real-Time Big Data Ingestion that Enables Continuous Processing

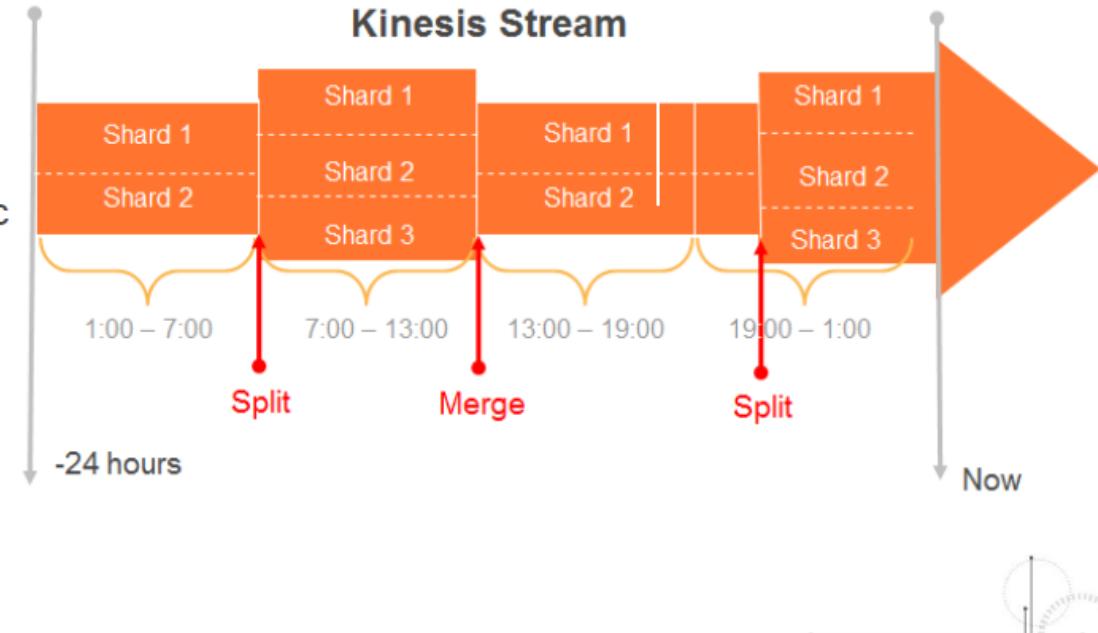


- Ensure record get processed
- Ensure records are distributed
- Match workers to shards..

Amazon Kinesis

Kinesis Stream: Managed ability to capture and store data

- Streams contain **Shards**
- Each Shard ingests data up to 1MB/sec, and up to 1000 TPS
- Each Shard emits up to 2 MB/sec
- All data is stored for **24 hours**
- **Scale** Kinesis streams by adding or removing Shards
- **Replay** data inside of 24Hr.



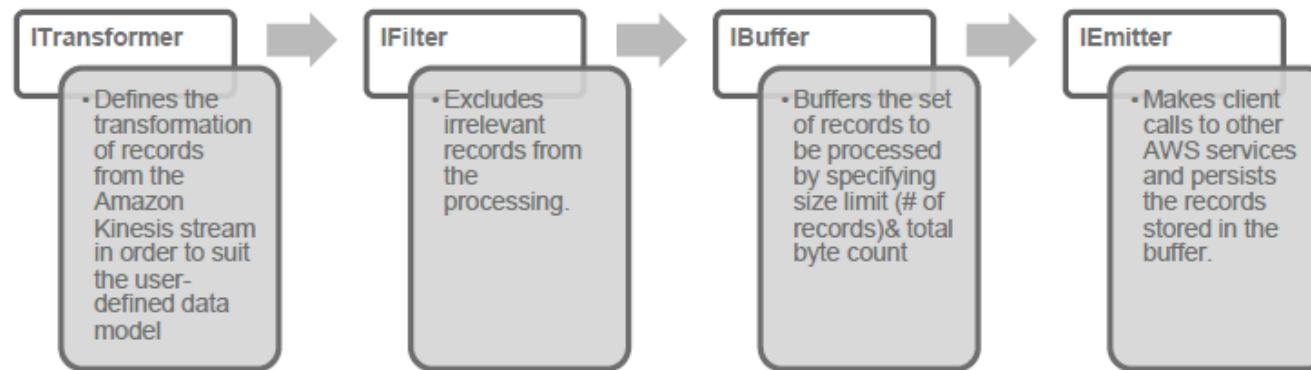
Amazon Kinesis

Amazon Kinesis Connector Library

Customizable, Open Source Apps to Connect Kinesis with S3, Redshift, DynamoDB



Kinesis



DynamoDB



Redshift

Amazon Kinesis

- Targeted towards special classes of applications
 - Built in connectors for several open data streams, e.g. Twitter
- Cloud deployment model
 - Streaming data to cloud has security, throughput limitations
- Micro-batch processing model
 - Performance and latency limitation
 - User control and programming flexibility limited
- More information
 - <http://aws.amazon.com/kinesis/>

Stream Processing Systems

Apache Spark and Spark Streaming

Objectives

- Introduction to Streaming Applications
 - Core development concepts
- Apache Spark, SparkSQL and Spark Streaming
 - Provide enough overview to get started
- Programming Hands On
 - <https://spark.apache.org/>

Anatomy of a Stream Processing System

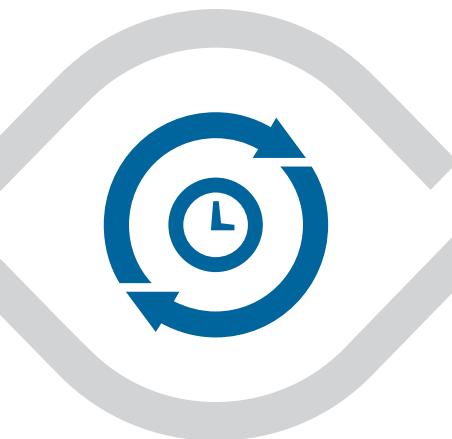
Integrated Development Environment

Scale-Out Runtime

Analytic Toolkits



Development and Management



Flexibility and Scalability



Functional and Optimized

Stream Processing Middleware Services

- Distribution
 - Plumbing related to moving processes around
- Resource management
 - Placement, scheduling, load shedding
- Job management
 - Starting/stopping/editing jobs, handling dynamic connections
- System monitoring
 - Health, state, performance of the system
- Security
- Logging
- Data Transport
 - Inter-Process, intra-Process, intra-host, inter-host
- Debugging
- Visualization

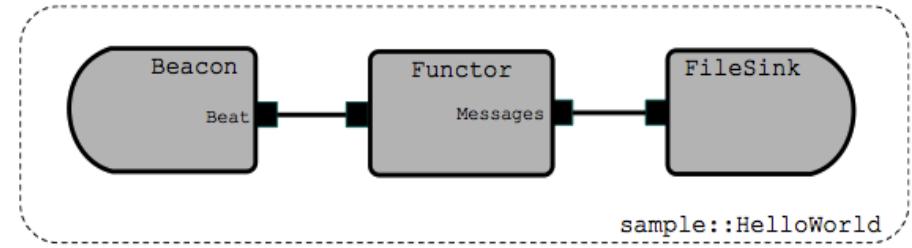
Stream Processing Applications

- Composition Language
- Defining Tuples and Streams
- Stream Sources
- Operators and Stream Transformers
- Windows and Stream Relational Processing
- Stream Sinks

Structure of a Streaming Application

SPL Hello World: A fully Declarative Model

```
1 namespace sample;
2
3 composite HelloWorld {
4     graph
5         stream<uint32 id> Beat = Beacon() {
6             param
7                 period: 1.0; // seconds
8                 iterations: 10u;
9             output
10                Beat: id = (uint32) (10.0*random());
11            }
12            stream<rstring greeting, uint32 id> Message = Functor(Beat) {
13                output
14                    Message: greeting = "Hello World! (" + (rstring) id + ")";
15                }
16                () as Sink = FileSink(Message) {
17                    param
18                        file: "/dev/stdout";
19                        format: txt;
20                        flush: 1u;
21                    }
22    }
```



Apache Spark and Spark Streaming

Apache Spark History

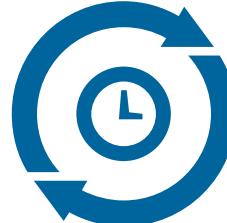
- Open-source in-memory cluster computing
- Developed at AMPLab
 - UC Berkeley (2009)
 - Open sourced in 2010
- First release: May 2014
- Overcomes several limitations of Hadoop Map-Reduce model

Apache Spark Platform

Development Environment

Scale-Out Runtime

Analytic Toolkits



Python, Scala, Java

In-Memory Cluster Computing

SparkMLlib, Connectors

Spark Components

Spark Streaming

Spark SQL

SparkMLlib

GraphX

Spark Core

Structured APIs: SQL, Dataframes and Datasets

Low-level APIs: RDDs and Distributed variables

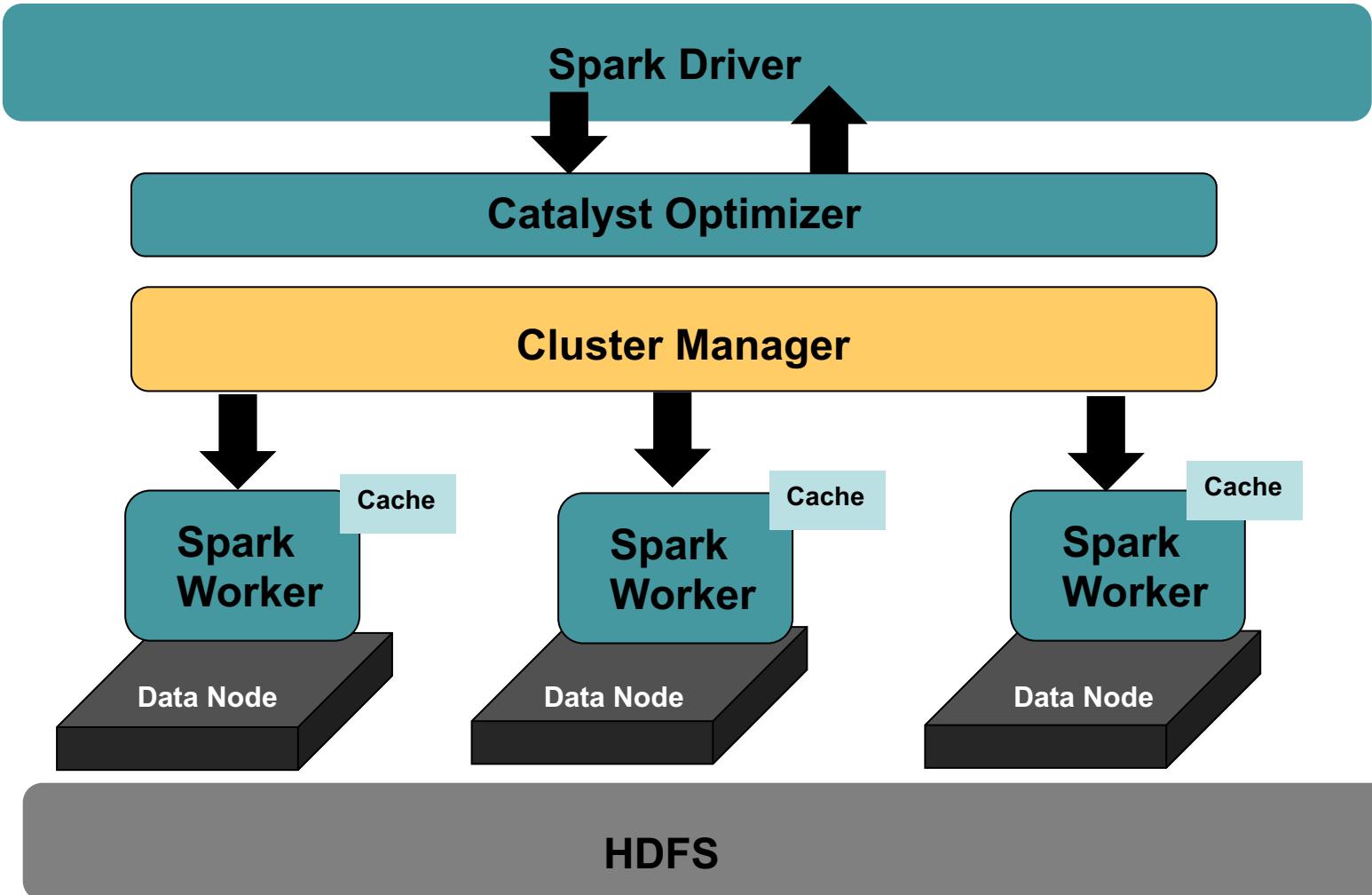
Cluster
Manager

Standalone
Scheduler

YARN

Mesos

Spark Runtime Architecture



Recall: Middleware Support

- Access transparency
 - Local and remote components accessible via same operations
- Location transparency
 - Components locatable via name, independent of location
- Concurrency transparency
 - Components/objects can be run in parallel
- Migration transparency
 - Allows movement of components without affecting other components
- Replication transparency
 - Allows multiple instances of objects for improved reliability (mirrored web pages)
- Failure transparency
 - Components designed while accounting for failure of other services

Spark Basics

- Installation
 - Straightforward on Linux/Windows
 - Download binaries – unzip
 - <http://spark.apache.org/downloads.html>
 - pip install pyspark
- Includes
 - Readme.md
 - bin (**binaries, including shell**)
 - core, streaming, python (**source code**)
 - examples

Spark Shell (Python and Scala)

- Shell
 - pyspark (Python)
 - spark-shell (Scala)
- Interactive command line based interface
- Similar to those in R, Python, Matlab
- Log level can be controlled
 - log4j.properties.template in conf folder
 - Copy to log4j.properties and set log level
 - log4j.rootCategory=WARN
- Easy to get started

Resilient Distributed Datasets (RDDs)

- Resilient
 - Data can be recreated if lost
- Distributed
 - Stored in memory across the cluster
- Dataset
 - Immutable
 - Lazy Evaluated
 - Cacheable
 - Type Inferred
- Fundamental unit of processing in Spark
 - Dstreams for Streaming

Immutability

- Data cannot be changed
 - Allows improved parallelism and scalability
 - However requires more storage

```
const int a = 0; //immutable
int b = 0; //mutable
b++; //allowed in-place change
a++; //not allowed
c = a+1;//create new variable - leave old variable intact
```

- Every transformation creates a new copy!

Lazy Evaluation

- Transformations are not computed till needed
 - Deferred evaluation
 - Separate execution from evaluation

```
C2 = C1.map(lambda val: val+1) //not evaluated  
C3 = C2.map(lambda val: val+2) //not evaluated  
print C3 //now evaluated as  
  
C3 = C1.map(lambda val : ((val+1)+2))
```

- Multiple transformations performed together when result needs to be reported
- Laziness works only when immutable data

Type Inference

- Type of a variable/collection inferred from values by compiler
- Sometimes can be inferred from transformation applied
 - Some transformations have fixed return type

```
C1 = [1, 2, 3, 4, 5] //Type array (explicit)
C2 = C1.map(lambda val: val+1) //C2 type array
C3 = C2.count() //Type integer
```

Caching

- Immutable data can be cached over long periods
- Lazy transformations allow for data to be recreated as needed
- Transformations can also be saved
- Caching improves performance

Programming with Spark

- Hands on Hello World

```
//read file line by line to create RDD called lines  
lines = sc.textFile("filename")  
//return count of lines in lines RDD  
lines.count()  
//return the first element of lines - first line in file  
lines.first()
```

- SC: Spark Context
 - Driver programs access Spark runtime through this context object
 - Represents a connection to the Spark Cluster
 - Created automatically in the shell

Programming with Spark

- Hands on Hello World

```
//read file line by line to create RDD called lines  
lines = sc.textFile("filename")  
//filter the RDD to extract lines with text test in them  
filteredLines = lines.filter(lambda val: "test" in val)  
//return count of lines in lines RDD  
lines.count()  
//return the first element of filteredLines  
filteredLines.first()
```

- filter
 - Special type of transformation to create a new RDD
 - Recall immutability and lazy transformations

Programming with Spark

- lambda Syntax
 - Allows for definition of inline functions in Python on Spark

```
//filter the RDD to extract lines with text test in them  
filteredLines = lines.filter(lambda val: "test" in val)
```

- Defines a boolean text filter in this case – that is applied to all the data items within the RDD. Can also be defined explicitly

```
//filter the RDD to extract lines with text test in them  
def findTest(x):  
    return "test" in x
```

```
filteredLines = lines.filter(findTest)
```

- Spark ships this function to all the executor nodes so that it can be run in parallel
- Equivalent => syntax for Scala

Standalone Programs with Spark

- Can invoke a program from outside the shell
 - Allows for ease of programming

```
//use spark-submit to submit custom python code  
spark-submit mycode.py
```

```
from pyspark import SparkConf, SparkContext  
sc = SparkContext("local", "myApp")  
lines = sc.textFile("foo")  
filteredLines = lines.filter(lambda x: "test" in x)
```

- In mycode.py “local” Master → single node deploy
 - Use a cluster URL that tells the driver how to connect to the cluster

Programming with Spark

- Two types of operations on RDDs
 - Transformations, Actions
- Transformations
 - Create new RDD by applying transformation to the RDD entries
- Actions
 - Compute a result based on an RDD and return to the driver program or write to output, e.g. HDFS
- Important distinction
 - Lazy evaluation model of Spark
 - Need for `persist` of RDDs: RDDs are recomputed every time an action is run on them. `persist` allows for caching of the result so that intermediate results can be reused

Programming with Spark

- RDD Creation

```
//parallelize method  
lines = sc.parallelize(["My", "test", "example here"])  
//read from file  
lines2 = sc.textFile("filename")
```

- RDD Transformations

```
//filter  
filteredLines = lines.filter(lambda x: "test" in x)  
filteredLines2 = lines.filter(lambda x: "example" in x)  
  
//union  
final = filteredLines.union(filteredLines2)
```

- What happens if we do not persist? What about lazy evaluation? ORs and ANDs?

Programming with Spark

- Element wise transformations
 - filter and map
- map
 - Apply stateless transform to each element within RDD

```
lines = sc.parallelize([1, 4, 3])
//read from file
lines2 = lines.map(lambda x: x*x)
res = lines2.collect()
for num in res:
    print num
```

- Element wise squaring of all the input values
- What does collect do?
 - Brings all the results back to the driver program as an array
 - Important to know – when RDD distributed across multiple nodes

Other Element Wise Transformations

- Element-wise one to many transformation
 - flatMap
 - Can create more (or less) than one element in output RDD per element in input RDD

```
//parallelize method  
lines = sc.parallelize(["My", "test", "example here"])  
//parse the text into words using space separator  
words = lines.flatMap(lambda x: x.split(" "))
```

- Third element in lines gets mapped to more than one element in the words RDD
- What is the difference from?

```
//parallelize method  
tokenLines = lines.map(lambda x: x.split(" "))
```

Set Operations on RDDs

- union
 - Set-wise union of two RDDs – retains duplicates
- distinct
 - Remove duplicate values in RDD
 - Expensive operation as needs to aggregate across all partitions of the RDD
- subtract
 - Set-wise difference between RDDs
 - Also removes duplicates → expensive operation
- intersection
 - Set-wise common elements between RDDs
 - Also removes duplicates

Set Operations on RDDs

```
lines = sc.parallelize(["coffee", "coffee", "panda", "monkey",
"tea"])
lines2 = sc.parallelize(["coffee", "monkey", "kitty"])

res1 = lines.union(lines2)
// {"coffee", "coffee", "coffee", "panda", "monkey", "monkey", "tea",
"kitty"}

res2 = lines.distinct()
// {"coffee", "panda", "monkey", "tea"}

res3 = lines.intersection(lines2)
// {"coffee", "monkey"}

res4 = lines.subtract(lines2)
// {"panda", "tea"}
```

Set Operations on RDDs

- cartesian operation
 - Create cartesian product of sets

```
lines = sc.parallelize(["coffee", "coffee", "panda"])
lines2 = sc.parallelize(["coffee", "monkey"])
```

```
res1 = lines.cartesian(lines2)
//{("coffee", "coffee"), ("coffee", "monkey"), ("coffee", "coffee"),
("coffee", "monkey"), ("panda", "coffee"), ("panda", "monkey")}
```

- Can be very expensive for large sets

Summary of Basic RDD Operations

```
lines = sc.parallelize([1, 2, 3, 3])
```

Name	Purpose	Example	Result
map()	Element-wise function, same number of output elements	lines.map(lambda x: x*x)	{1, 4, 9, 9}
flatMap()	Element-wise function, different number of output elements	lines.flatMap(lambda x: [0,x])	{0,1,0,2,0,3,0,3}
filter()	Select RDD elements that satisfy a condition	lines.filter(lambda x: x>1)	{2, 3, 3}
distinct()	Remove duplicates - expensive	lines.distinct()	{1, 2, 3}
sample(withReplacement, fraction, [seed])	Randomly sample from RDD	lines.sample(false, 0.5)	{1,3} Non-deterministic

Summary of Two-RDD Operations

```
a = sc.parallelize([1, 2, 3])
```

```
b = sc.parallelize([3, 4, 5])
```

Name	Purpose	Example	Result
union()	Union across all elements – duplicates retained	a.union(b)	{1, 2, 3, 3, 4, 5}
subtract()	Set-difference – duplicates removed	a.subtract(b)	{1, 2}
intersection()	Common elements – duplicates removed	a.intersection(b)	{3}
cartesian()	Cartesian product - expensive	a.cartesian(b)	{(1,3), (1,4), (1,5), (2, 3), (2,4), (2,5), (3,3), 3,4, (3,5)}

Actions on RDDs: Aggregations

- Produce result of running an operation on RDD
 - not new RDD
 - Often recursive application of some function
- `reduce` **action**
 - Takes function that operates on two elements on RDD and returns element of same type
 - Applies function recursively across all elements in RDD

```
nums = sc.parallelize([1, 2, 3, 4])
res = nums.reduce(lambda x, y: x+y)
//returns sum of entire RDD = 10
```

- Note that `res` is not a RDD – it is an integer value

Actions on RDDs: Aggregations

- Similar to `reduce` is the `fold` action
 - Except we can add an initial value for the result

```
nums = sc.parallelize([1, 2, 3, 4])
res = nums.reduce(lambda x, y: x+y)
//returns sum of entire RDD = 10
res2 = nums.fold((5), (lambda x,y:x+y))
//returns the sum starting with an initial value 5 = 15
```

- More complex action is `aggregate`
 - Frees the constraint on the function that return value has to be same as the element value
 - Takes initial value, function to combine elements into accumulator, and another function to combine accumulators

Actions on RDDs: Aggregations

- Example to compute sum and count at same time

- Initialize with 0,0

```
nums = sc.parallelize([1, 2, 3, 4])
res = nums.aggregate(
    (0,0),
    (lambda acc, val: (acc[0]+val, acc[1]+1)),
    (lambda acc1, acc2: (acc1[0]+ acc2[0], acc1[1] + acc2[1])))
//returns sum and count = (10,4)
//can compute average
print res[0]/res[1]
```

- First function sums RDD values and increments counter by 1
 - Next function combines the accumulators – potentially across multiple partitions of the RDD
- Can we do something equivalent with map and reduce functions only?

Actions on RDDs: Collection

- The `collect` action
 - Collects all values across the RDD across all partitions and workers and brings them back as a list to driver program (can be expensive for a large RDD)
 - Expects the resulting list to fit in memory
- The `take(n)` action
 - Takes n values from the RDD and returns to the driver program as a list
 - Order of return values may vary based on optimization across partitions
- The `top` action
 - Return top value based on a specified order
- The `takeSample(withReplacement, num, seed)` action
 - Return num samples using random sampling – with or without replacement

Actions on RDDs

```
lines = sc.parallelize([1, 2, 3, 3])
```

Name	Purpose	Example	Result
collect()	Collect all the RDD elements into a list	lines.collect()	{1, 2, 3, 3}
count()	Count elements in RDD	lines.count()	4
countByValue()	Count based on value of RDD – expensive action	lines.countByValue	{(1,1),(2,1),(3, 2)}
take(num)	Collect num elements into a list	lines.take(2)	{1, 2}
top(num)	Collect num elements based on order	lines.top(2)	{3,3}

Actions on RDDs

```
lines = sc.parallelize([1, 2, 3, 3])
```

Name	Purpose	Example	Result
takeOrdered(num, (ordering))	Collect num elements based on order function	lines.takeOrdered(2, (myorderingfunc))	{1, 2}
takeSample(withReplacement, num, [seed])	Get elements in RDD	lines.takeSample(false, 2)	Non-deterministic
reduce(func)	Recursive aggregation of RDD based on function	lines.reduce(lambda x, y: x+y)	9
fold(init, func)	Recursive aggregation of RDD based on function with init val	lines.fold(2, (lambda x, y: x+y))	11
Aggregate(init, seqOp, combOp)	Recursive aggregation that allows combination function with return type different than RDD element type	See earlier example	
foreach(func)	Apply function for each element of RDD	lines.foreach(func)	Depends on func

RDD Persistence

- Due to lazy evaluation
 - RDDs are processed only when action applied
 - If multiple actions applied to same RDD – it may be computed multiple times
 - Persistence allows for caching of intermediate results

```
nums = sc.textFile("foo")
res = nums.top()
res2 = nums.count()
//File read two times
```

```
nums = sc.textFile("foo")
nums.persist()
res = nums.top()
res2 = nums.count()
//File read only once
```

- Each node stores its computed partition for reuse
- If a node fails, the partition is recomputed
- Old partitions evicted using a Least Recently Used (LRU) policy

Pair RDDs

- RDDs that have key,value elements
 - Common to a majority of data analysis tasks
 - Expose new operations and partitioning
 - Can be read from certain data formats
 - Created using the map function

```
lines = sc.textFile("foo")
pairlines = lines.map(lambda x: (x.split(" ")[0], x))
```

- Taking the first word in a line as the key to the line
- This returns tuple elements within the resulting RDD
- Have additional transformation functions that can be applied

Pair RDD Transformations

Rdd: { (1,2), (3,4), (3, 6) } lines: { ("a","This is"), ("b", "a test") }

Name	Purpose	Example	Result
groupByKey ()	Group elements in RDD by key to make element lists in new RDD	Rdd.groupByKey ()	{(1,[2]),(3,[4,6])}
mapValues (func)	Apply function to each value without changing the key	Rdd.mapValues (lambda x: x*x)	{(1,4),(3,16),(3,36)}
flatMapValues (func)	Apply function to each RDD element without changing the key – each function can return more than one value	lines.flatMapValues (lambda x:x.split (""))	{"a","This"}, {"a","is"}, {"b","a"}, {"b","test"}
keys ()	Return an RDD with just the keys	Rdd.keys ()	{1,3,3}

Pair RDD Transformations

Rdd: { (1, 2), (3, 4), (3, 6) } other: { (3, 9) }

Name	Purpose	Example	Result
values ()	Return an RDD with just the values	Rdd.values ()	{2,4,6}
sortByKey ()	Create new RDD sorted by key	Rdd.sortByKey ()	{(1,2),(3,4), (3,6)}
subtractByKey ()	Set operation - remove elements with key common to second RDD . Removes duplicates	Rdd.subtractByKey (other)	{(1,2)}
cogroup ()	Group elements across two RDDs with the same key. Also does a groupByKey within each RDD	Rdd.cogroup (other)	{(1,[2],[]), (3,[4,6],[9])}

Join: A Database Perspective

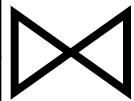
- An SQL join on tables in a relational database :

```
SELECT NAME, EQUIP  
      FROM EMP-DEPT JOIN DEPT-EQUIP ON  
EMP-DEPT.DEPT = DEPT_EQUIP.DEPT
```

- Inner Join:* Result contains only tuples that were matched

EMP-DEPT

Name	Dept.
Alice	Accounting
Bob	Programming
Carol	Sales
Dan	HR
Ellen	Programming
Frank	Accounting
George	Research



DEPT-EQP

Dept.	Equipment
Accounting	Calculator
Programming	Computer
Sales	Computer
HR	Tablet
Security	Vehicle
Sales	Phone



Name	Equipment
Alice	Calculator
Bob	Computer
Ellen	Computer
Dan	Tablet
Carol	Computer
Carol	Phone
Frank	Calculator

Join: A Database Perspective

- An SQL join on tables in a relational database :

```
SELECT NAME, EQUIP
```

```
FROM EMP-DEPT LEFT OUTER JOIN DEPT-EQUIP ON  
EMP-DEPT.DEPT = DEPT_EQUIP.DEPT
```

- Left Outer Join:* Result contains tuples that were matched, and unmatched tuples from the “left” table

EMP-DEPT

Name	Dept.
Alice	Accounting
Bob	Programming
Carol	Sales
Dan	HR
Ellen	Programming
Frank	Accounting
George	Research



DEPT-EQP

Dept.	Equipment
Accounting	Calculator
Programming	Computer
Sales	Computer
HR	Tablet
Security	Vehicle
Sales	Phone



Name	Equipment
Alice	Calculator
Bob	Computer
Ellen	Computer
Dan	Tablet
Carol	Computer
Carol	Phone
Frank	Calculator
George	NULL

Join: A Database Perspective

- An SQL join on tables in a relational database :

```
SELECT NAME, EQUIP
```

```
    FROM EMP-DEPT RIGHT OUTER JOIN DEPT-EQUIP ON  
    EMP-DEPT.DEPT = DEPT_EQUIP.DEPT
```

- Right Outer Join:* Result contains tuples that were matched, and unmatched tuples from the “right” table

EMP-DEPT

Name	Dept.
Alice	Accounting
Bob	Programming
Carol	Sales
Dan	HR
Ellen	Programming
Frank	Accounting
George	Research



DEPT-EQP

Dept.	Equipment
Accounting	Calculator
Programming	Computer
Sales	Computer
HR	Tablet
Security	Vehicle
Sales	Phone



Name	Equipment
Alice	Calculator
Bob	Computer
Ellen	Computer
Dan	Tablet
Carol	Computer
Carol	Phone
Frank	Calculator
NULL	Vehicle

Aside: Additional Joins in Spark

- Semi-joins: Left and Right

Only return rows from the left (or right) table that match, and no duplicates

EMP-DEPT

Name	Dept.
Alice	Accounting
Bob	Programming
Carol	Sales
Dan	HR
Ellen	Programming
Frank	Accounting
George	Research



DEPT-EQP

Dept.	Equipment
Accounting	Calculator
Programming	Computer
Sales	Computer
HR	Tablet
Security	Vehicle
Sales	Phone



Name	Equipment
Alice	Accounting
Bob	Programming
Ellen	Programming
Dan	HR
Carol	Sales
Frank	Accounting

- Anti-joins: Apply a not to a Semi-join
 - Left and Right

Joins on Pair RDDs

- Joins operate like database counterparts
 - Inner, LeftOuter and RightOuter
 - Equality conditions are defined based on keys

```
Rdd: { (1, 2), (3, 4), (3, 6) } other: { (3, 9) }
```

```
Rdd.join(other): { (3, (4, 9)), (3, (6, 9)) }
```

```
Rdd.leftOuterJoin(other): { ((1, (2, None)), (3, (4, 9))),  
                           (3, (6, Some(9))) }
```

```
Rdd.rightOuterJoin(other): { (3, (4, 9)), (3, (6, 9)) }
```

- Is Join a symmetric operation?
- How about RightOuterJoin and LeftOuterJoin?
- How about non-equi joins?

Aggregations on Pair RDDs

- reduce, fold **and** aggregate actions extended to work by key
 - reduceByKey, foldByKey, aggregateByKey
 - These operate as transformations rather than as actions, i.e. return RDDs, since each RDD may have multiple keys

Rdd: { (1,2), (3,4), (3, 6) } other: { (3,9) }

```
Res = Rdd.reduceByKey(lambda x,y:x+y)
//Res = { (1,2), (3,10) }
```

```
Res2 = Rdd.foldByKey(3,lambda x,y:x+y)
//Res2 = { (1,5), (3,13) }
```

```
Res3 = Rdd.mapValues(lambda x: (x,1)).reduceByKey(lambda x,y:
(x[0]+y[0],x[1]+y[1]))
```

- What does the above do?

WordCount Example on Spark

- WordCount
 - Count occurrences of each word within a file

```
lines = sc.textFile("foo")
words = lines.flatMap(lambda x: x.split(" "))
wordMap = words.map(lambda x: (x,1))
wordCount = wordMap.reduceByKey(lambda x,y: x+y)

wordCount2 = words.countByKey()

wordCount3 = words.count()
```

- What is the difference in these different results?

Partitioning and Parallelism

- How does Spark parallelize the processing?
 - Every RDD has a fixed number of partitions
 - When grouping or aggregating RDDs, we can specify degree of partitions
 - Especially true and useful for pair RDDs

```
Data = sc.parallelize([(“a”,1), (“b”,2), (“c”,3)])  
Res = Data.reduceByKey(lambda x,y: x+y)  
//default parallelism  
Res = Data.reduceByKey(lambda x,y: x+y, 10)  
//Custom parallelism use 10 partitions
```

- Sometimes data may need to be repartitioned to improve efficiency of the operation – for groupings

Partitioning and Parallelism

- Optimize partitioning to minimize communication across nodes
 - Especially useful when a dataset is used multiple times
- Spark does not allow explicit control on which worker node a key goes to
 - Can specify which sets of keys go together
 - E.g. hash partition by key mod 100 → all keys with the same remainder after hashing to 100 will be on same node
 - Range partition based on key values – so that keys in same range appear on same node

```
Data = sc.parallelize([(“a”,1), (“b”,2), (“c”,3)])
Res2 = Data.partitionBy(new HashPartitioner(100)).persist()
```

- Scala example
- Custom partitioners can be written in Scala and Java

Other Actions on Pair RDDs

Rdd: { (1, 2), (3, 4), (3, 6) } other: { (3, 9) }

Name	Purpose	Example	Result
collectAsMap()	Collect all results as a map and return to driver program	Rdd.collectAsMap()	Map object
countByKey()	Count number of elements for each key	Rdd.countByKey()	{(1,1),(3,2)}
lookup()	Return values that have a matching key as an array	Rdd.lookup(3)	[4,6]

Next

- Spark Structured APIs and Dataframes

Structured Spark: Dataframes

- Part of the structured API to handle and process data
 - Equivalent to a table with named and typed columns (i.e. with a schema)
 - Just that the table is distributed over multiple workers
- Analogous to Pandas Dataframes

```
linesdf =  
spark.read.option("inferSchema","true").option("header","true")  
 .csv("data/structured_data.csv")  
  
linesdf.first()
```

- Same notions of lazy execution and actions (like for RDD) carry over

Transformations/Actions on Dataframes

- What do you think this does?

```
from pyspark.sql.functions import desc  
  
linesdf.groupBy("Line").count().withColumnRenamed("count", "n_products").sort(desc("n_products")).show(5)
```

Transformations on Dataframes

```
df1 = src, dest, flights
```

```
NY , CA , 100
```

```
CA , CA , 25
```

```
CA , NY , 100
```

Name	Purpose	Example	Result
select()	Select columns in dataframe	df1.select("src")	{ NY, CA, CA }
selectExpr()	Select and apply transforms and aggregations (can create new cols or dfs)	df1.selectExpr("*", "(src = dest) as inState")	NY, CA, 100, False CA, CA, 25, True CA, NY, 100, False
withColumn()	Explicitly create new columns – can use literals (as constants)	df1.withColumn("const", lit(50))	src, dest, flights, const NY, CA, 100, 50 CA, CA, 25, 50 CA, NY, 100, 50

Transformations on Dataframes

```
df1 = src, dest, flights
```

```
NY , CA , 100
```

```
CA , CA , 25
```

```
CA , NY , 100
```

Name	Purpose	Example	Result
drop()	Drop columns	df1.drop("src")	dest, flights CA, 25 NY, 100 CA, 100
where()	Filter rows	df1.where("flights<50")	CA, CA, 25
distinct()	Unique rows	df1.select("src").distinct()	NY CA
sort()/order By()	Sort based on some columns	df1.orderBy("src","flights")	src, dest, flights CA, CA, 25 CA, NY, 100 NY, CA, 100
union()	Append dataframe rows with the same schema	df1.union(df1)	

Transformations on Dataframes

```
df1 = src, dest, flights      def manyFlights(x):  
    NY , CA , 100                return x<50  
    CA , CA , 25  
    CA , NY , 100
```

Name	Purpose	Example	Result
udf()	Register a user defined function	manyFlightsUdf = udf(manyFlights)	
	Can use this UDF with select or selectExpr	df1.select(manyFlightsUdf(col("flights"))))	manyFlights(flights) False True False

Two Dataframe Transforms

df1 = src, dest, flights

NY , CA , 100

CA , CA , 25

CA , NY , 100

df2 = state, pop

NY , 100

CA , 200

Name	Purpose	Example	Result
join()	Applies join across dataframes	<pre>je = (df1.col("src") == df2.col("state")) df1.join(df2, je)</pre>	src, dest, flights, state, pop CA, CA, 25, CA, 200 CA, NY, 100, CA, 200 NY, CA, 100, NY, 100
	Support different types of joins		

Actions on Dataframes

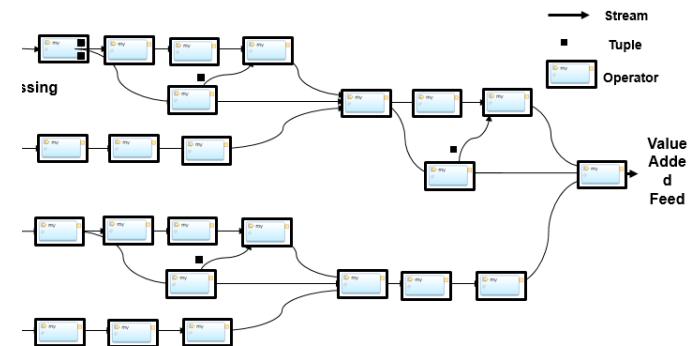
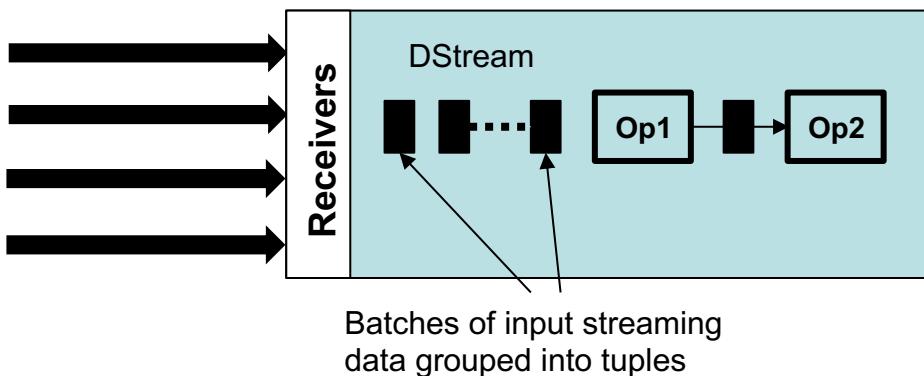
Name	Purpose	Example	Result
count()	Count the number of occurrences	df.select(count("src"))	3
countDistinct()	Count number of distinct occurrences	df.select(countDistinct("src"))	2
avg(), sum(), min(), max()	Apply aggregates to column	df.select(max("flights"))	100
approx_count_distinct()	Get approximate distinct counts		
show()	Show a few rows of the dataframe	df1.show()	
collect()	Collect and bring to local memory – show metadata and content	df1.collect()	
take(n)	Take n rows from dataframe – show metadata and content, but flatten	df1.take(5)	

Next

- Spark Streaming

Spark Streaming

- Introduce concept Dstream (discretized streams)
 - Sequence of RDDs arriving over time
 - Dstreams can be created from different data sources
 - Network interfaces, Flame, Kafka, Flume, File System etc.
 - Support transformations and actions (output operations)



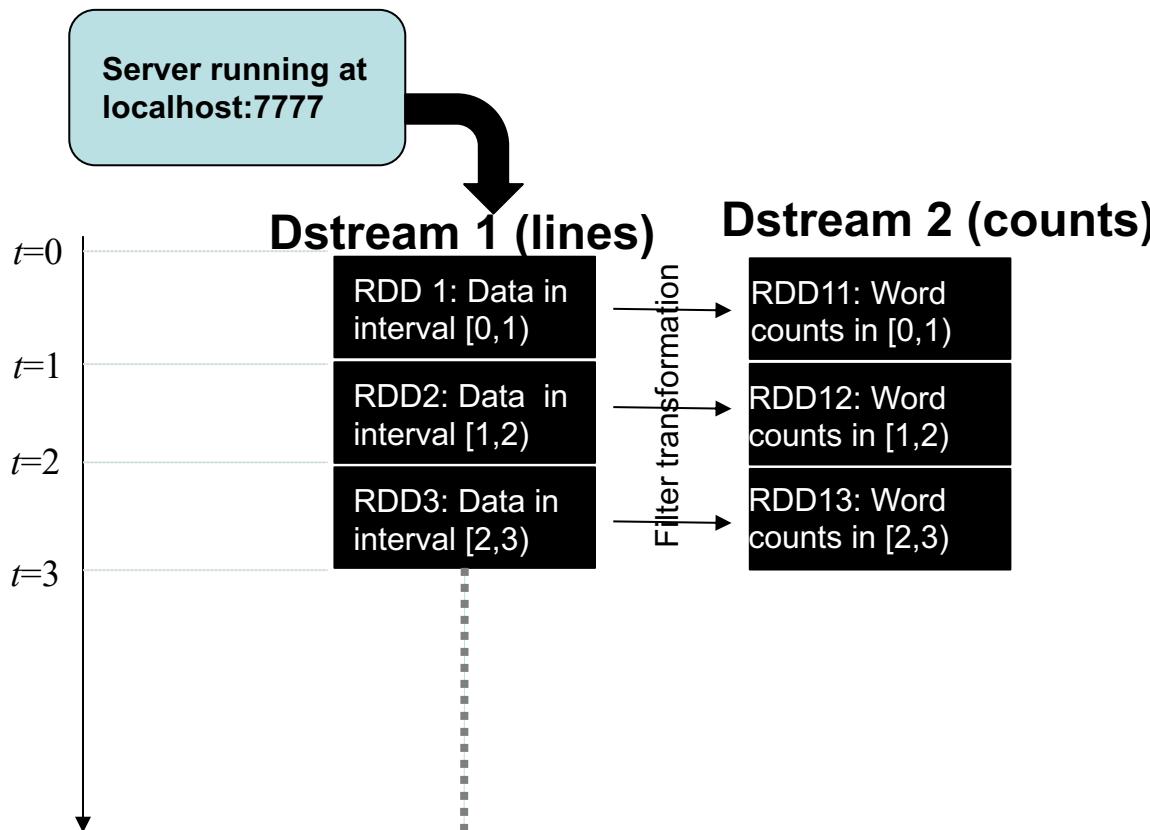
Streaming Example

- Example of constructing Dstream from network socket
 - Setup a Dstream

```
sc = SparkContext(appName="StreamingNetworkWordCount")
ssc = StreamingContext(sc, 1)
lines = ssc.socketTextStream(IP, Port)
counts = lines.flatMap(lambda line: line.split(" ")) \
              .map(lambda word: (word, 1)) \
              .reduceByKey(lambda a, b: a+b)
counts.pprint()
ssc.start()
ssc.awaitTermination
```

- Take all data received in one second
- Break it into words and then do a wordcount
- Print all results to screen

Streaming Example



The operation of this word count is “stateless” – each window handled independently

Stateless Streaming Transformations

Name	Purpose	Example
map()	Apply function defined inline to each tuple (RDD) in the stream	ds.map(lambda x: x+1)
flatMap()	Apply inline function to each tuple in stream and create tuple with more than one element per input element	ds.flatMap(lambda x: x.split(" "))
filter()	Filter elements in input tuple to create new tuple	ds.filter(lambda x: x.contains("error"))
reduceByKey()	Perform reduce by key within tuple to create new tuple	ds.reduceByKey(lambda (x,y): x+y)
groupByKey()	Group values by key within each tuple	ds.groupByKey()
transform()	Apply any RDD to RDD function on each tuple	ds.transform(lambda rdd: myFunc(rdd))

Stateless Two-Stream Transformations

- Stateless Join on two Streams
 - Perform a standard RDD join on two tuples, one from each stream
 - Fast stream needs to wait for slow stream

```
//Create a streaming context with a 1 second batch
sc = SparkContext(appName="StreamingStatelessJoin")
ssc = StreamingContext(sc, 1)
ipDstream = accessLogs.map(lambda x: (x.getIpAddress(),1))
ipCountDstream = ipDstream.reduceByKey(lambda (x,y): x+y)

ipByteDstream = accessLogs.map(lambda x: (x.getIpAddress(),x.getContentSize()))
ipByteCountDstream = ipByteDstream.reduceByKey(lambda (x,y): (x+y))

ipBCDstream = ipByteCountDstream.join(ipCountDstream)
ssc.start()
ssc.awaitTermination()
```

This can cause memory issues if one stream is much slower than the other
Also – limiting in terms of match conditions, given data arrival order etc,

Stateful Transformation on Streams

- Create and maintain state as tuples processed
- Such state, along with internal algorithm, affects the results
- e.g. DeDuplicate
 - *tuple is considered a duplicate if it shares the same key with a previously seen tuple within a pre-defined period of time*
- Runtime support is challenging
 - Require synchronization in a multi-threaded context
 - No trivial way to parallelize
 - Require some persistence mechanism for fault-tolerance

Windowing

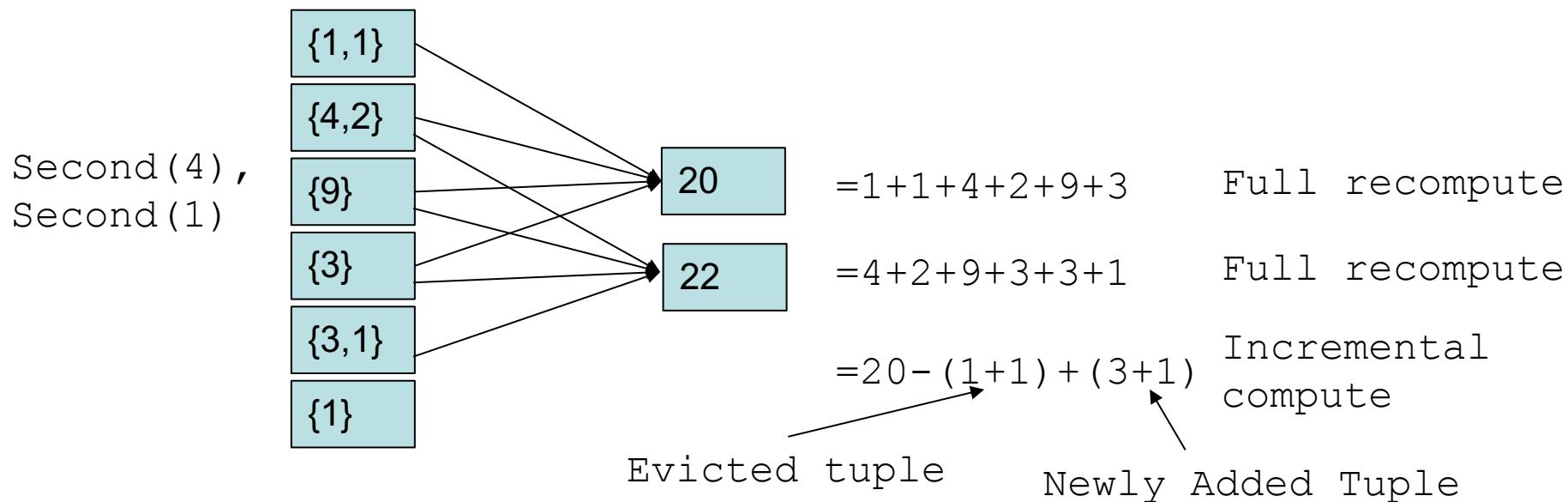
- Window creates a collection of tuples/RDDs within window
- Has size, slide – corresponds to eviction, trigger policies

```
//Create a streaming window  
winLogs = accessLogs.window(30, 10)  
winCounts = winLogs.countByKey()
```

- Slide cannot be smaller than batch size of Dstream
- Performs a Union across RDDs in the window
- Can then apply a reduce across this to compute additional results
 - More efficient aggregations available

Windowed Aggregations

- Incremental operations for aggregations
 - `reduceByWindow`, `reduceByKeyAndWindow`
 - Instead of re-computing fully, remove effect of evicted tuples and add in effect of newly added tuples
 - Require easily reversible functions, e.g. sum



Windowed Aggregations

```
//Create a streaming reduce with a 1 second batch, 4 second window
ipDstream = accessLogs.map(lambda x: (x.getIpAddress(),1))
ipCountDstream = ipDstream.reduceByKeyAndWindow(
    lambda (x,y): x+y,
    4,
    1)
ipCountEDstream = ipDstream.reduceByKeyAndWindow(
    lambda (x,y): x+y,
    lambda (x,y): x-y,
    4,
    1)
```

Reversible function specification followed by window specification allows incremental computation, as opposed to full recompute
Other functions include `reduceByWindow()`, `countByWindow()` and `countByKeyAndWindow()`

Spark Streaming: Inputs and Outputs

- `print()` – grabs first 10 elements from each tuple and prints them to screen
- `saveAsTextFiles("output", "txt")` – writes multiple files into a directory, one per tuple in txt format
- Stream of files
 - `val logData = ssc.textFileStream("logDirectory")`
 - Each file read as a tuple in the Dstream
- Other sources
 - Twitter
 - Kafka, Kinesis, Flume, and others

Limitations of Spark Streaming Model

- All windows defined in wall-clock time
 - Sizes and slides limited to multiples of batch size
- No data-time windows
 - Skewed by randomness in data arrival
 - Cannot handle out of order data
- Limited join and aggregation conditions

Next

- Structured Spark Streaming

References

- H. Karau, A. Konwinski, P. Wendell and M. Zaharia, “Learning Spark”, O'Reilly Press
- <http://spark.apache.org/>
- Chapter 4 in Stream Processing Book

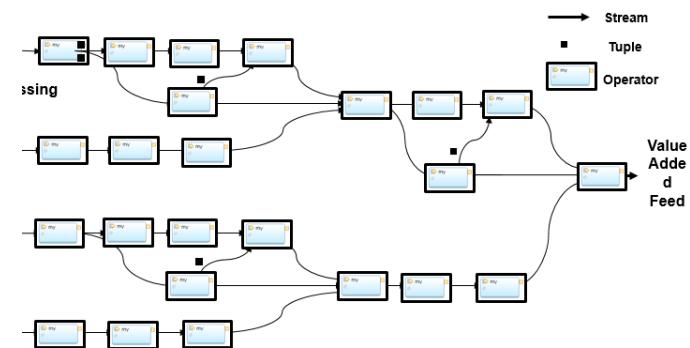
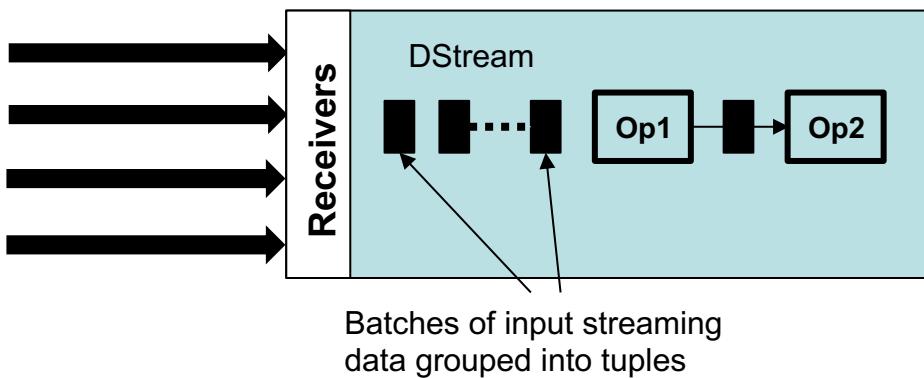
Spark Streaming and Apache Beam

Objectives

- Apache Spark Streaming
 - Programming
- Apache Beam Hands On
 - Programming
- HW1

Spark Streaming

- Introduce concept Dstream (discretized streams)
 - Sequence of RDDs arriving over time
 - Dstreams can be created from different data sources
 - Network interfaces, Flame, Kafka, Flume, File System etc.
 - Support transformations and actions (output operations)



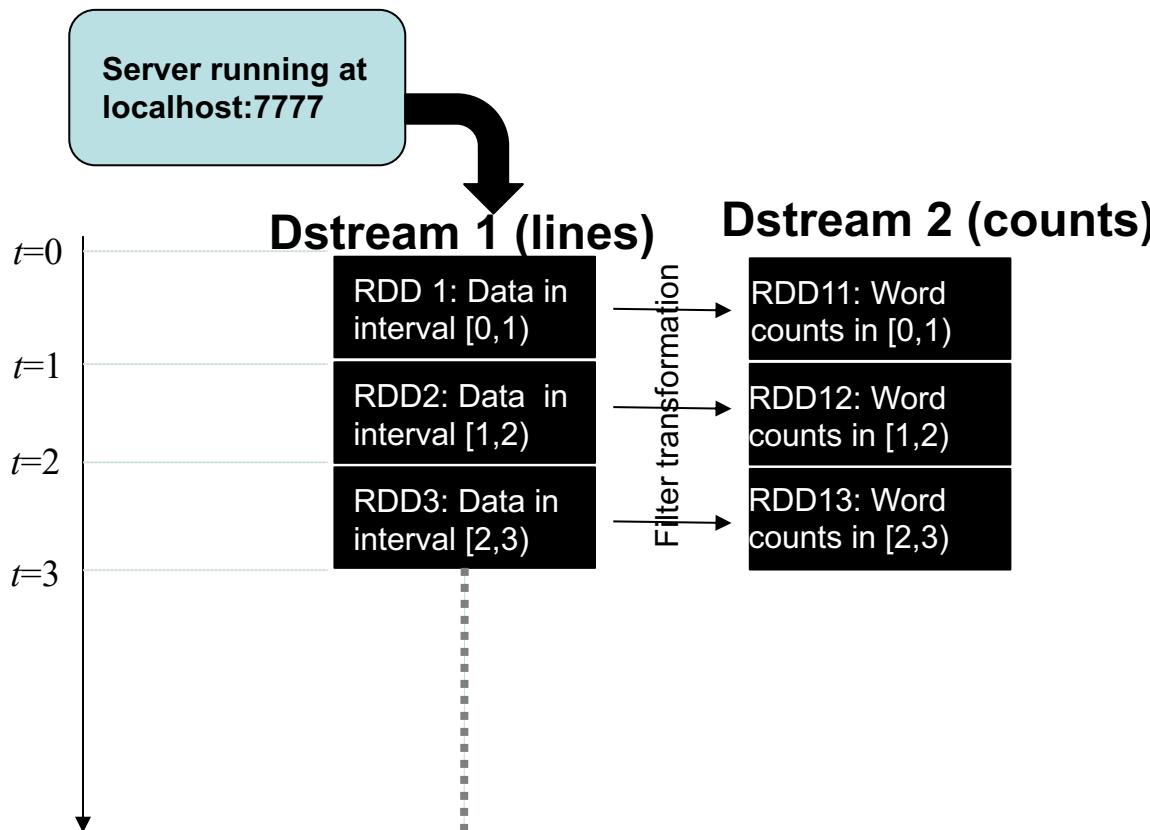
Streaming Example

- Example of constructing Dstream from network socket
 - Setup a Dstream

```
sc = SparkContext(appName="StreamingNetworkWordCount")
ssc = StreamingContext(sc, 1)
lines = ssc.socketTextStream(IP, Port)
counts = lines.flatMap(lambda line: line.split(" ")) \
              .map(lambda word: (word, 1)) \
              .reduceByKey(lambda a, b: a+b)
counts.pprint()
ssc.start()
ssc.awaitTermination
```

- Take all data received in one second
- Break it into words and then do a wordcount
- Print all results to screen

Streaming Example



The operation of this word count is “stateless” – each window handled independently

Stateless Streaming Transformations

Name	Purpose	Example
map()	Apply function defined inline to each tuple (RDD) in the stream	ds.map(lambda x: x+1)
flatMap()	Apply inline function to each tuple in stream and create tuple with more than one element per input element	ds.flatMap(lambda x: x.split(" "))
filter()	Filter elements in input tuple to create new tuple	ds.filter(lambda x: x.contains("error"))
reduceByKey()	Perform reduce by key within tuple to create new tuple	ds.reduceByKey(lambda (x,y): x+y)
groupByKey()	Group values by key within each tuple	ds.groupByKey()
transform()	Apply any RDD to RDD function on each tuple	ds.transform(lambda rdd: myFunc(rdd))

Stateless Two-Stream Transformations

- Stateless Join on two Streams
 - Perform a standard RDD join on two tuples, one from each stream
 - Fast stream needs to wait for slow stream

```
//Create a streaming context with a 1 second batch
sc = SparkContext(appName="StreamingStatelessJoin")
ssc = StreamingContext(sc, 1)
accessLogs = ssc.socketTextStream(IP, Port)
ipDstream = accessLogs.map(lambda x: (x.getIpAddress(),1))
ipCountDstream = ipDstream.reduceByKey(lambda (x,y): x+y)

ipByteDstream = accessLogs.map(lambda x: (x.getIpAddress(),x.getContentSize()))
ipByteCountDstream = ipByteDstream.reduceByKey(lambda (x,y): (x+y))

ipBCDstream = ipByteCountDstream.join(ipCountDstream)
ssc.start()
ssc.awaitTermination()
```

This can cause memory issues if one stream is much slower than the other
Also – limiting in terms of match conditions, given data arrival order etc,

Stateful Transformation on Streams

- Create and maintain state as tuples processed
- Such state, along with internal algorithm, affects the results
- e.g. DeDuplicate
 - *tuple is considered a duplicate if it shares the same key with a previously seen tuple within a pre-defined period of time*
- Runtime support is challenging
 - Require synchronization in a multi-threaded context
 - No trivial way to parallelize
 - Require some persistence mechanism for fault-tolerance

Aside: Data Time versus Wall Clock Time

- Both are representations of time
 - With different references
- Data Time
 - Event Time, data timestamp
- Wall Clock Time
 - Arrival time, Processing time

Windows and Stateful Processing

- Sorting, aggregating, or joining data in a relational table
 - All data in the table can be processed
- For streaming data, data flows continuously
 - No beginning, no end
 - Requires a different paradigm for sorting, aggregating and joining
 - Can only work with a subset of consecutive tuples
- This finite set of tuples is called a window
 - Note that in Spark – each tuple is a RDD (collection of elements)

Window Properties

- Three properties that define a window
 - Eviction policy
 - Defines how large a window can get
 - Determines which older tuples are removed from the window
 - Trigger policy
 - When an operation, such as aggregation, takes place as new tuples arrive into the window
 - Partitioning
 - Maintains separate windows for each group of tuples with the same grouping key value
- Window types
 - Tumbling windows:
 - Non-overlapping sets of consecutive tuples
 - Defined only with an eviction policy
 - Sliding windows:
 - Windows formed by adding new tuples to the end of the window and evicting old tuples from the beginning of the window
 - Defined with both an eviction and a trigger policy

Policy Specifications

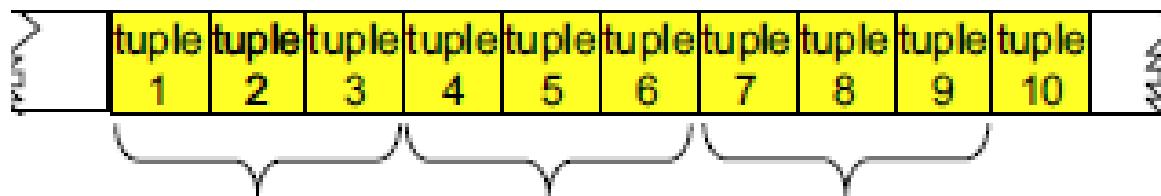
- Count (Fixed window size)
 - a fixed number set of tuples
- Time (Fixed processing time interval)
 - set of tuples that arrived in a specified period of time
- Delta (event time based on specified numeric or timestamp attribute)
 - a set of tuples where each of the tuple's specified attribute value is no more than x greater than that of the first tuple in the window
 - Often used with timestamps that are part of tuple data
 - Ideally expects monotonic increase in attribute value across tuples
- Punctuation
 - a set of tuples that are between punctuations (control tuples)

Tumbling Window

- Is specified by providing an eviction policy only
 - Punctuation
 - Count
 - Time
 - Attribute delta
- Stores tuples until the window is full
 - based upon the eviction policy
- When the window is full
 - Executes the operator behavior
- After the behavior has been executed
 - Flushes the window

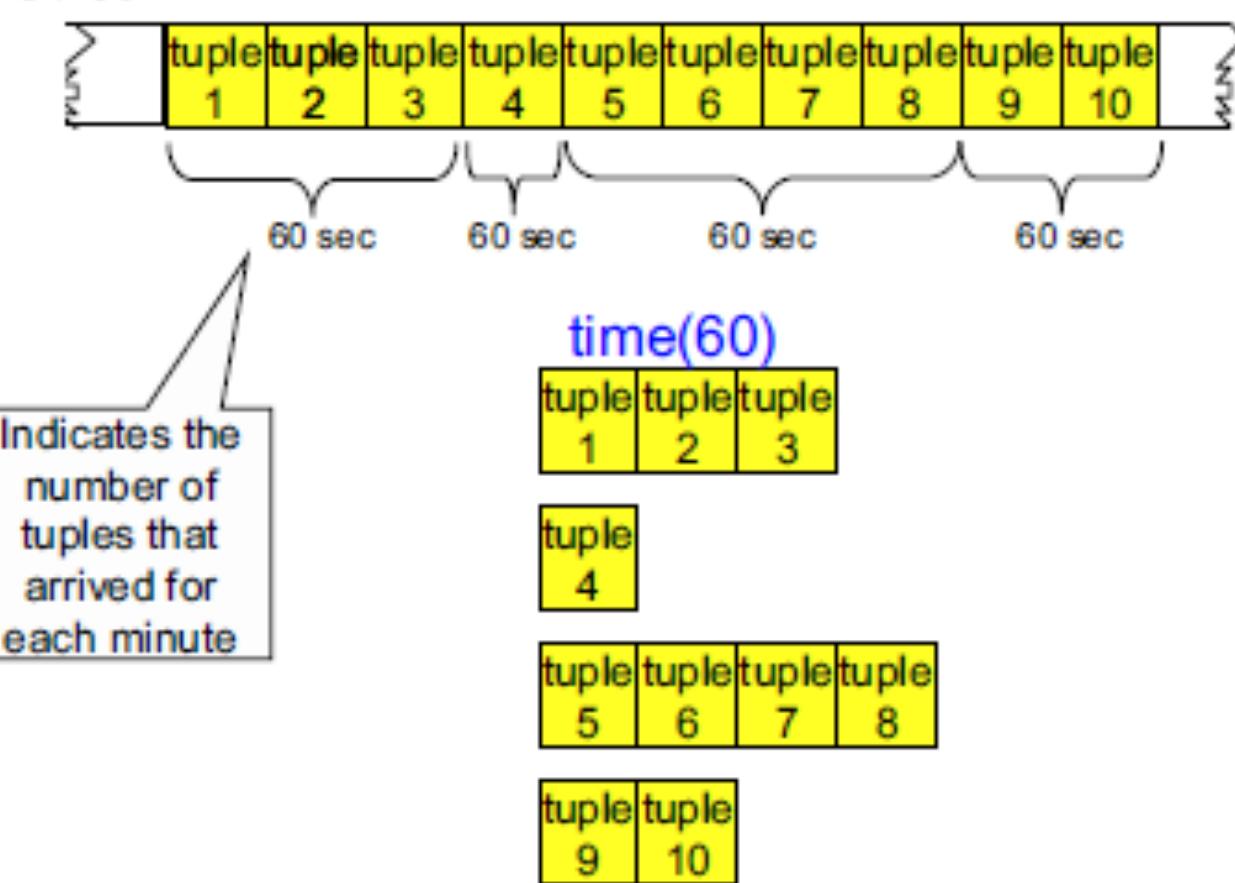
Count Based Tumbling Windows

Stream:



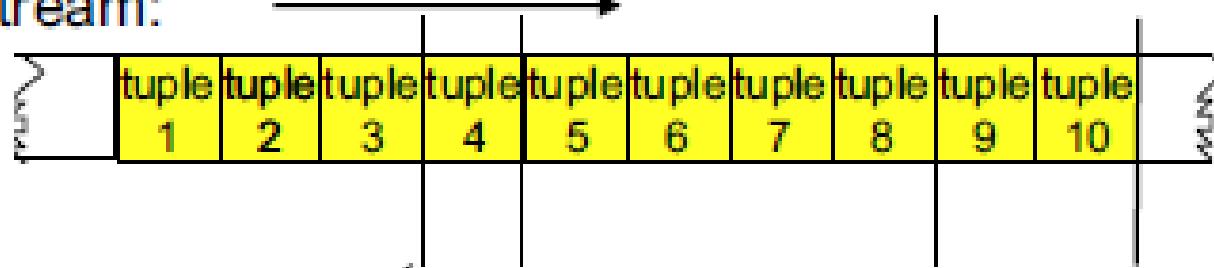
Time Based Tumbling Windows

Stream:



Punctuation Based Tumbling Windows

Stream:



Indicates a
punctuation
mark

tuple	tuple	tuple
1	2	3

tuple
4

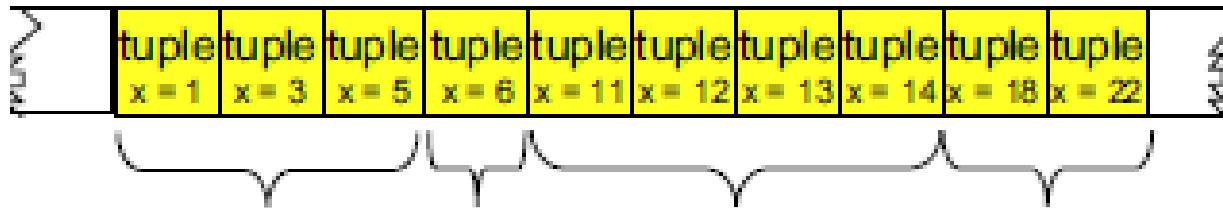
tuple	tuple	tuple	tuple
5	6	7	8

tuple	tuple
9	10

...

Delta Based Tumbling Windows

Stream:



Indicates the group of tuples for which the first and last tuple's designated attribute is within a delta amount

$\text{delta}(x, 4)$

tuple	tuple	tuple
x = 1	x = 3	x = 5

tuple
x = 6

tuple	tuple	tuple	tuple
x = 11	x = 12	x = 13	x = 14

tuple	tuple
x = 18	x = 22

...

Sliding Windows – Eviction Policies

- When a new tuple arrives
 - It is added to the end of the window
 - Zero or more old tuples are evicted from the start of the window (first-in/first-out)
- Three alternatives for determining how many tuples are evicted:
 - Count
 - Once window fills up, one old tuple evicted for each arriving tuple
 - Time
 - Enough tuples evicted to maintain a maximum age
 - No remaining tuple is more than the given period older than the newly arriving tuple
 - Expired tuples are not removed when the time period is elapsed, but when a new tuple arrives
 - Attribute delta
 - Evict tuples for which a specified attribute's value is more than *delta* less than that attribute's value in new tuple

Sliding Windows – Trigger Policies

- Three alternatives for determining when a sliding window is processed:
 - `count(n)`
 - When a specified number of tuples have arrived since the last time a window was processed
 - `time(n)`
 - When a specified amount of time has passed
 - `delta(attrib, value)`
 - When a tuple arrives whose specified non-decreasing numeric attribute value is more than a specified amount greater than that attribute's value for the tuple that triggered the last processing operation
- Eviction policy and trigger policy are independent
 - In general, all nine combinations are possible
 - Spark supports two combinations – with same trigger and eviction policy
 - Beam supports multiple combinations –Java more than Python

Sliding Window

Stream:



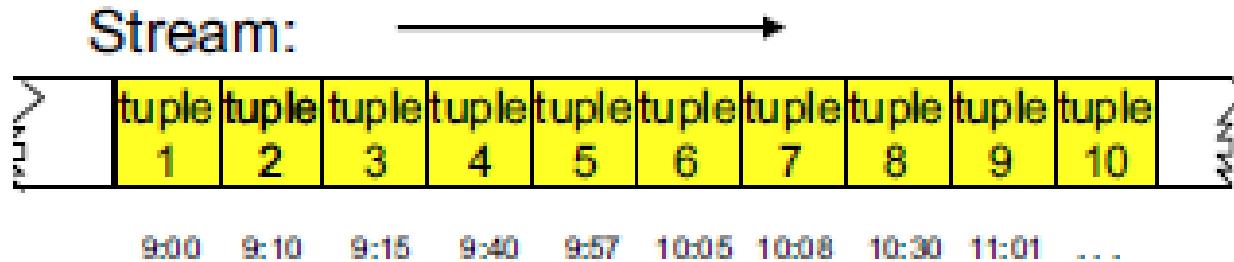
count(5), count(2)



No trigger



Sliding Window



time(60), count(1)

tuple	tuple	tuple	tuple	tuple
1	2	3	4	5

tuple	tuple	tuple	tuple	tuple
2	3	4	5	6

tuple	tuple	tuple	tuple	tuple	tuple
2	3	4	5	6	7

tuple	tuple	tuple	tuple	tuple
4	5	6	7	8

Sliding Window

Stream:



$\text{delta}(x, 4), \text{count}(1)$

tuple	tuple	tuple
x=1	x=3	x=5

tuple	tuple	tuple
x=3	x=5	x=6

tuple
x=11

tuple	tuple
x=11	x=12

tuple	tuple	tuple
x=11	x=12	x=13

Windowing

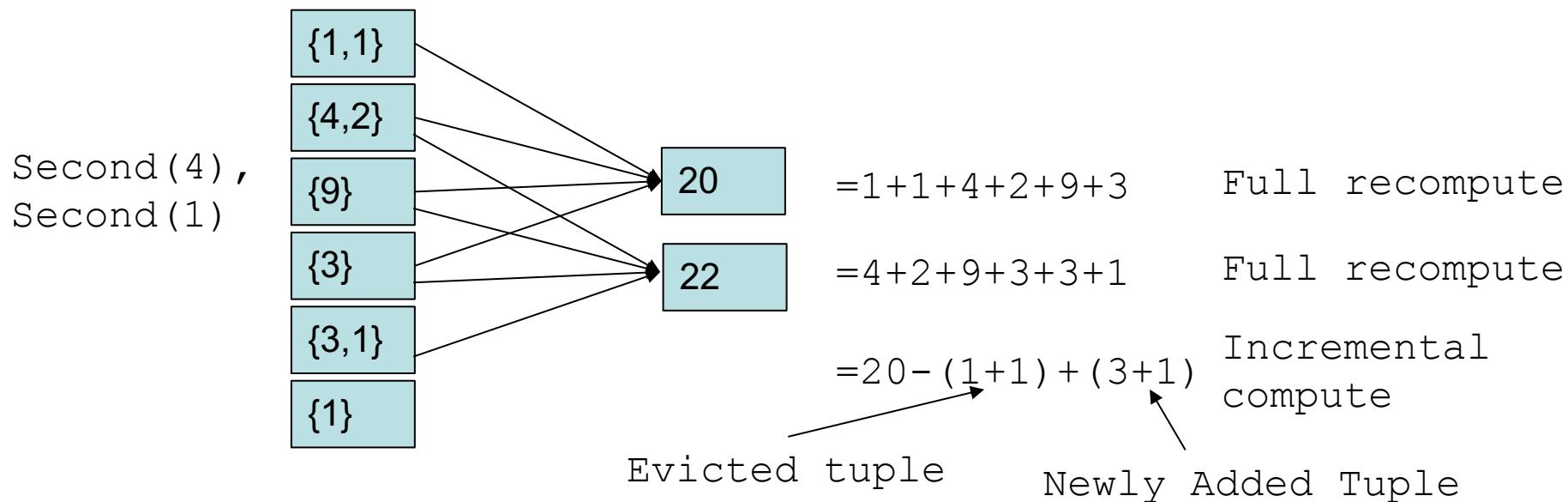
- Window creates a collection of tuples/RDDs within window
- Has size, slide – corresponds to eviction, trigger policies

```
//Create a streaming window  
winLogs = accessLogs.window(30, 10)  
winCounts = winLogs.countByKey()
```

- Slide cannot be smaller than batch size of Dstream
- Performs a Union across RDDs in the window
- Can then apply a reduce across this to compute additional results
 - More efficient aggregations available

Windowed Aggregations

- Incremental operations for aggregations
 - `reduceByWindow`, `reduceByKeyAndWindow`
 - Instead of re-computing fully, remove effect of evicted tuples and add in effect of newly added tuples
 - Require easily reversible functions, e.g. sum



Windowed Aggregations

```
//Create a streaming reduce with a 1 second batch, 4 second window
ipDstream = accessLogs.map(lambda x: (x.getIpAddress(),1))
ipCountDstream = ipDstream.reduceByKeyAndWindow(
    lambda (x,y): x+y,
    4,
    1)
ipCountEDstream = ipDstream.reduceByKeyAndWindow(
    lambda (x,y): x+y,
    lambda (x,y): x-y,
    4,
    1)
```

Reversible function specification followed by window specification allows incremental computation, as opposed to full recompute
Other functions include `reduceByWindow()`, `countByWindow()` and `countByKeyAndWindow()`

Spark Streaming: Inputs and Outputs

- `print()` – grabs first 10 elements from each tuple and prints them to screen
- `saveAsTextFiles("output", "txt")` – writes multiple files into a directory, one per tuple in txt format
- Stream of files
 - `val logData = ssc.textFileStream("logDirectory")`
 - Each file read as a tuple in the Dstream
- Other sources
 - Twitter
 - Kafka, Kinesis, Flume, and others

Limitations of Spark Streaming Model

- All windows defined in wall-clock time
 - Sizes and slides limited to multiples of batch size
- No data-time windows
 - Skewed by randomness in data arrival
 - Cannot handle out of order data
- Limited join and aggregation conditions

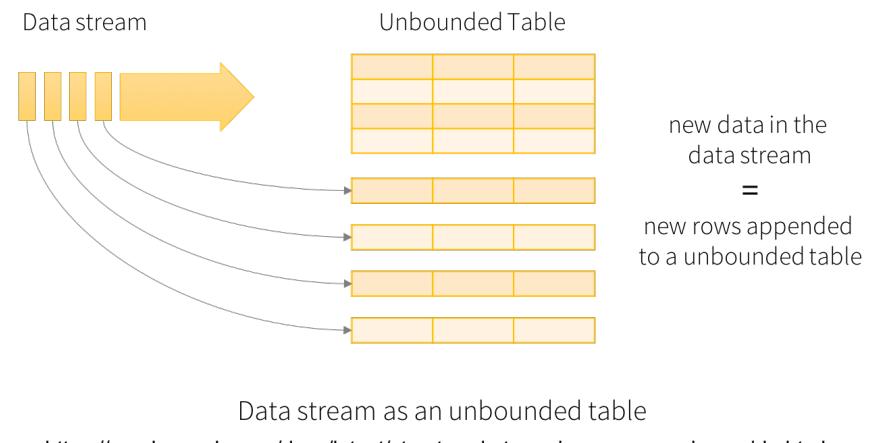
Spark Structured Stream Programming

- Uses SQL-like commands on Dataframes
 - Built on top of the Spark SQL engine
- Structured streaming queries processed as micro-batches (like Dstreams)
- Word Count Example (from before)

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import explode
from pyspark.sql.functions import split
spark = SparkSession.builder.appName \
    ("StructuredNetworkWordCount").getOrCreate()
lines = spark.readStream.format("socket").option("host","localhost").\
    option("port", 9999).load()
words = lines.select(explode( split(lines.value, " ")).alias("word") )
wordCounts = words.groupBy("word").count()
```

Structured Stream Programming

- Different from the concept of “true” streaming
 - Each received line viewed as a row in continuously growing table (with column called “value”)



- Can write results out periodically (1 sec by default)

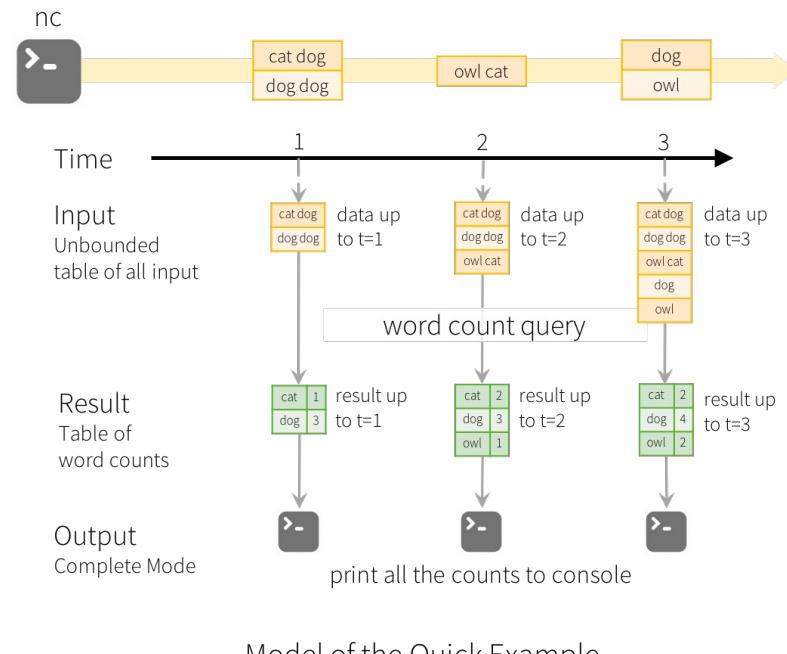
```
query = wordCounts.writeStream.outputMode("complete") \  
       .format("console").start()
```

```
query.awaitTermination()
```

- Keeps query running in background

Writing Results Periodically

- *Complete Mode* - Entire updated Result Table written.
- *Append Mode* - Only results of new rows written.
- *Update Mode* - Only results for all updated rows mode



Model of the Quick Example

<https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html>

- Does not materialize entire table – instead computes results from previous results, incrementally

Structured Streaming Operations

- All operations are SQL queries
 - Selection, projection, joins, aggregates etc.
 - Can create streaming dataframe from multiple sources

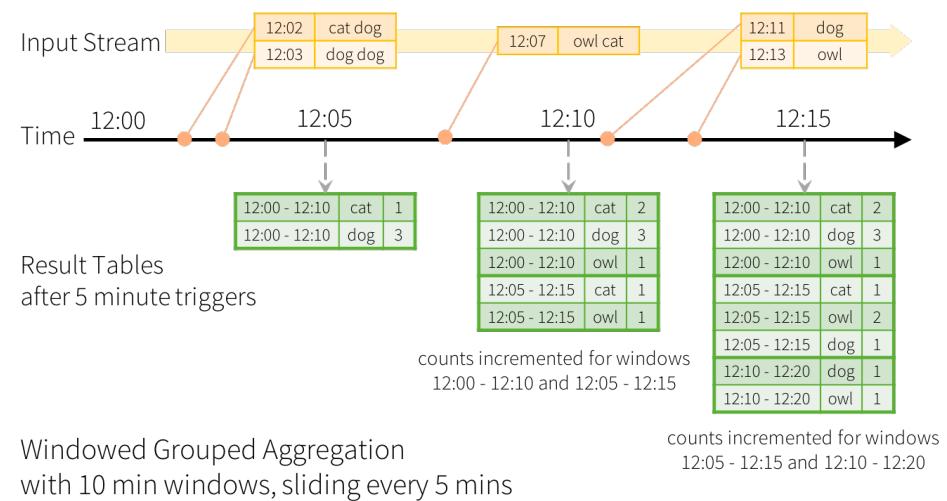
```
userSchema = StructType().add("device", "string").add("deviceType",  
"string").add("signal", "double").add("time", "DateType")  
  
csvDF = spark.readStream.option("sep",  
";").schema(userSchema).csv("/path/to/directory")  
  
#streaming DataFrame with schema { device: string, deviceType: string, signal:  
double, time: DateType }  
  
gt10df = csvDF.select("device").where("signal > 10")  
  
counts = csvDF.groupBy("deviceType").count()
```

Structured Streaming: Event Windows

- Support both time based windows as well as **event time** (data time) based windows

```
words = ... # streaming DataFrame of
schema { timestamp: Timestamp, word:
String }
# Group the data by window and word and
compute the count of each group
```

```
windowedCounts = words.groupBy(
window(words.timestamp, "10 minutes",
"5 minutes"), words.word ).count()
```

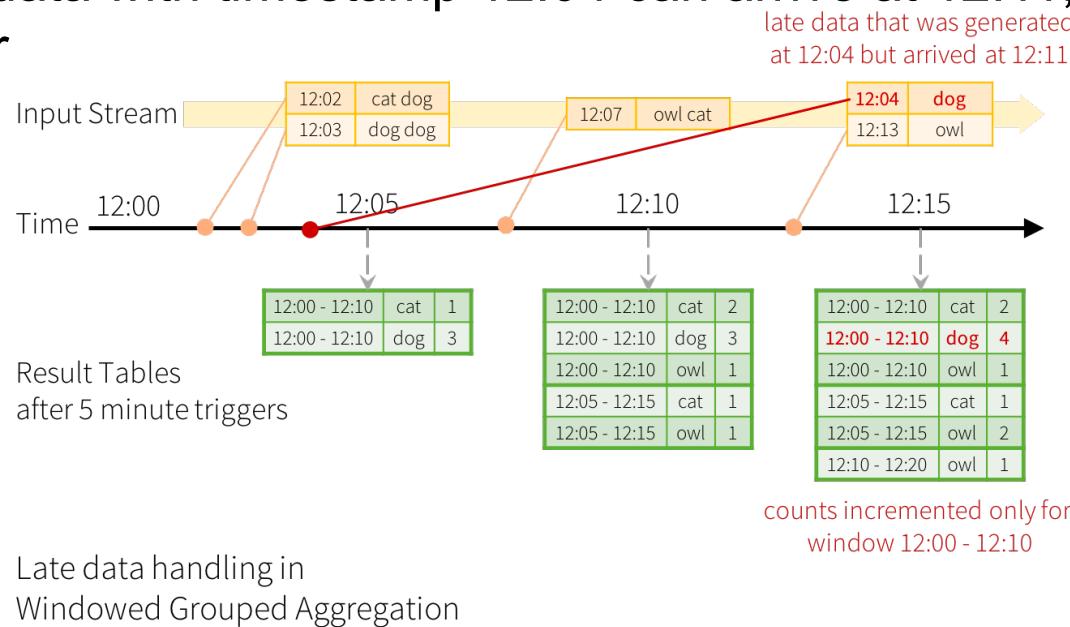


<https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html>

- This is useful when data time and arrival time are different due to real-world issues
- Recall : Triggers are at micro-batch intervals

Structured Streaming: Watermarking

- Watermark: specifies maximum delay to wait for late arriving data
 - e.g. data with timestamp 12:04 can arrive at 12:11, or out of order



<https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html>

- Without watermark, need to maintain unbounded state

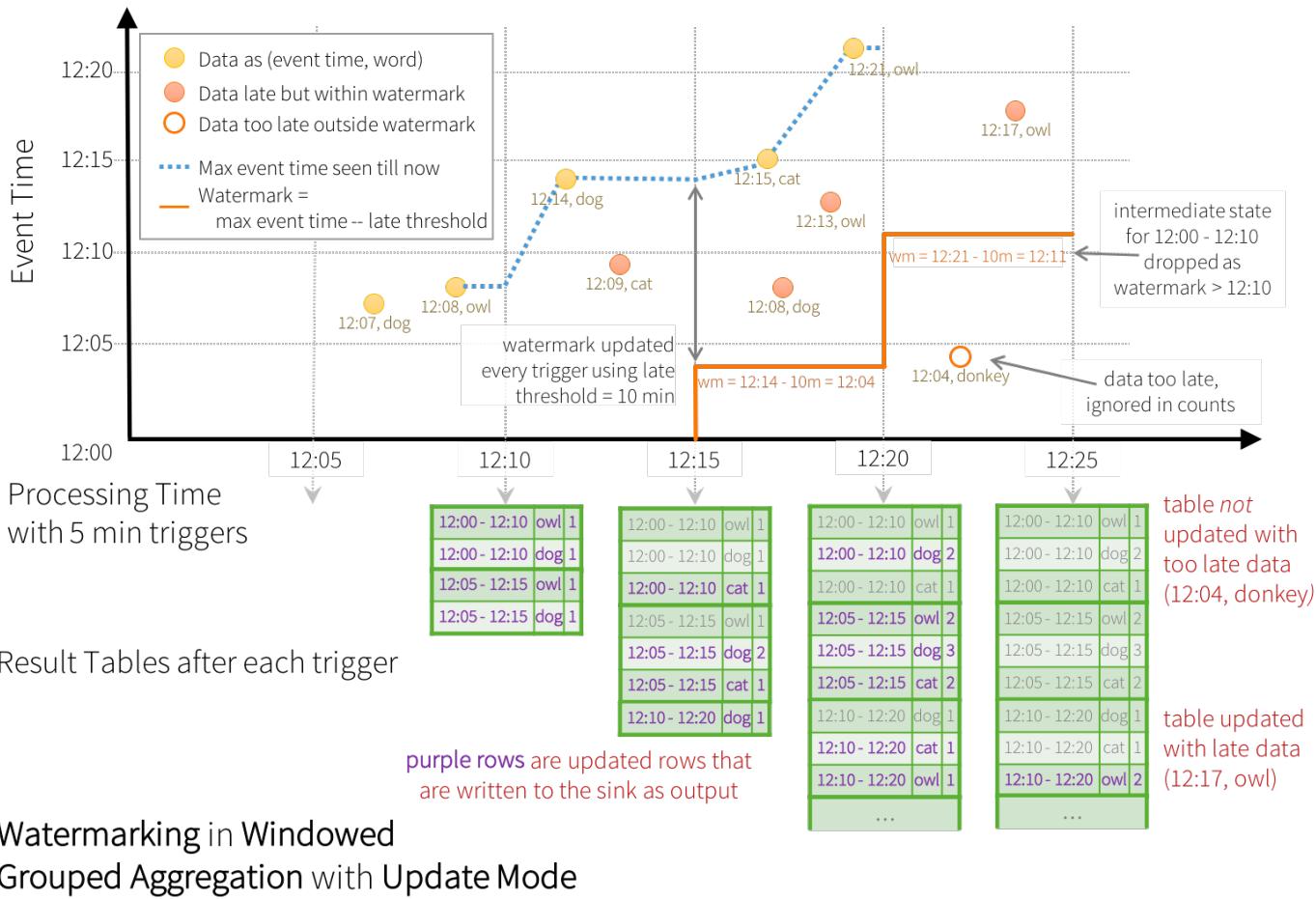
Structured Streaming: Watermarking

- Allows system to know when old data can be dropped
 - Do not need to update aggregates anymore
- Watermark specified in terms of event time
 - Defines maximum out of order in the data (not really lateness)

```
words = ... # streaming DataFrame schema { timestamp: Timestamp, word: String }
# Group the data by window and word and compute the count of each group
windowedCounts = words.withWatermark("timestamp", "10 minutes") \
    .groupByKey(words.timestamp, "10 minutes", "5 minutes"), words.word) \
    .count()
```

- Data that is older than the watermark (10 min) is discarded
- Output mode must be Append or Update

Structured Streaming: Watermarking



<https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html>

Watermarking and Other Processing

Join

```
impressions = spark.readStream....  
clicks = spark.readStream....  
# Apply watermarks on event-time columns impressionsWithWatermark =  
impressions.withWatermark("impressionTime", "2 hours")  
clicksWithWatermark = clicks.withWatermark("clickTime", "3 hours")  
# Join with event-time constraints  
impressionsWithWatermark.join( clicksWithWatermark, expr(""" clickAdId =  
impressionAdId AND clickTime >= impressionTime AND clickTime <=  
impressionTime + interval 1 hour """) )
```

Deduplicate

```
streamingDf = spark.readStream. ....  
# With watermark using guid and eventTime columns  
streamingDf.withWatermark("eventTime", "10 seconds") \  
.dropDuplicates("guid", "eventTime")
```

- When joining streams with multiple watermarks, a global watermark is computed as the watermark of the slowest stream – so that no data is accidentally deleted

Structured Streaming: Input and Output

```
# Read text from socket
socketDF = spark.readStream.format("socket") \
.option("host", "localhost").option("port", 9999).load()

# Read all the csv files written atomically in a directory
userSchema = StructType().add("name", "string").add("age", "integer")
csvDF = spark.readStream.option("sep", ";") \
.schema(userSchema).csv("/path/to/directory")

# Write to file/s
writeStream.format("parquet") // can be "orc", "json", "csv", etc.
.option("path", "path/to/destination/dir") .start()

# Write to Kafka
writeStream.format("kafka").option("kafka.bootstrap.servers",
"host1:port1,host2:port2").option("topic", "updates") .start()

# Write to Console
writeStream .format("console") .start()
```

Other Issues

- Recommended minimum latency is 500msec
 - Receivers can have too much overhead with less than that
- Scaling
 - Multiple receivers run in parallel, and union can be used to merge streams
 - Need to be careful about ordering
 - Can repartition the data
- Fault tolerance
 - Checkpointing based fault tolerance

References

- <https://spark.apache.org/docs/latest/streaming-programming-guide.html>
- <https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html>
- H. Karau, A. Konwinski, P. Wendell and M. Zaharia, “Learning Spark”, O’Reilly Press
- <http://spark.apache.org/>
- Chapter 4 in Stream Processing Book

Apache Beam

Apache Beam

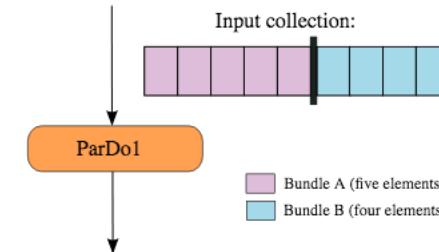
- Open source unified model
 - Composition language for data pipelines
 - For batch and streaming jobs
- Multiple language SDK
 - Java, Python, Go, Scala
- Multiple execution frameworks (runners)
 - Apache Apex 
 - Apache Flink 
 - Apache Gearpump 
 - Apache Samza 
 - Apache Spark 
 - Google Dataflow 
 - IBM Streams

Apache Beam Runners Support

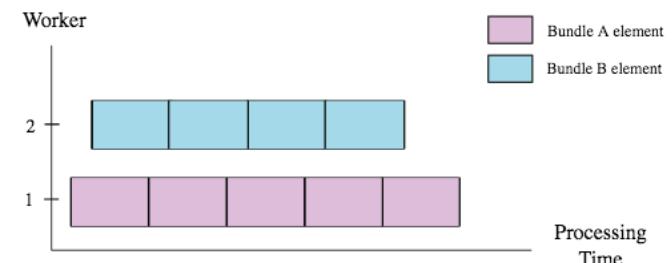
- Access transparency
 - Local and remote components accessible via same operations
- Location transparency
 - Components locatable via name, independent of location
- Concurrency transparency
 - Components/objects can be run in parallel
- Migration transparency
 - Allows movement of components without affecting other components
- Replication transparency
 - Allows multiple instances of objects for improved reliability (mirrored web pages)
- Failure transparency
 - Components designed while accounting for failure of other services
- Different runners implement these differently, however some common requirements

Runners: Access, Location, Concurrency

- **Serialization and Communication**
 - Shipping data items from one pipeline stage to another (core function of distributed system)
 - Done for grouping (as reduce), for parallelization, or for persistence
 - Uses either disk, network or in-memory (for transforms on the same node)
- **Bundling and Parallelization**
 - Focus on data parallel tasks (see later)
 - Each data stream (Pcollection) can be decomposed into ***bundles*** and distributed to multiple workers



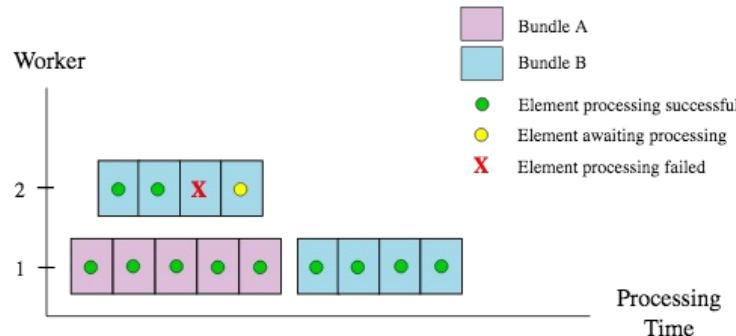
<https://beam.apache.org/documentation/execution-model/>



<https://beam.apache.org/documentation/execution-model/>

Runners: Failure Tolerance

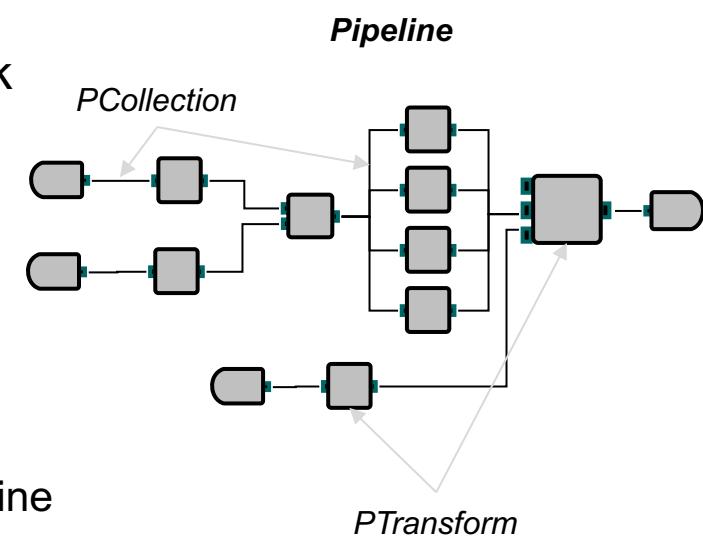
- Provided at the granularity of ***bundles***
 - If any element in the bundle fails, processing recomputed



- Note that bundle may be processed on a different node than the original processing
- Guarantees at least once processing
- What is the difference between batch processing and stream processing?

Apache Beam Programming

- Define driver program using one of Beam SDK languages
 - Java, Python, Go
- Core abstractions
 - *Pipeline* (application flowgraph)
 - Encapsulates entire data processing task
 - Needs to specify where and how to run
 - *PCollection* (Dataset or Stream)
 - Bounded datasets for batch processing
 - Unbounded for stream processing
 - *PTransform* (Operator)
 - Data processing operation or step in the pipeline
 - Multiple special I/O transforms for PCollections



Apache Beam Programming: Pipeline

Creating a pipeline in Python

```
import apache_beam as beam
from apache_beam.options.pipeline_options import PipelineOptions

def mypipelinerun():
    p = beam.Pipeline(options=PipelineOptions())
```

Creating a pipeline in Java

```
import org.apache.beam.sdk.Pipeline;
import org.apache.beam.sdk.options.PipelineOptions;
import org.apache.beam.sdk.options.PipelineOptionsFactory;

public static void main(String[] args) throws IOException {
    PipelineOptions options = PipelineOptionsFactory.create();
    // Then create the pipeline.
    Pipeline p = Pipeline.create(options);
}
```

Pipeline Options: Information about runner to use, project info (e.g. cloud account) etc.
Allows reading these options from command line as --<option>=<value> at submission time

Pipelines and Pipeline Options

- Can define custom options to be passed at submission time

```
class MyOptions(PipelineOptions):  
    @classmethod def _add_argparse_args(cls, parser):  
        parser.add_argument('--input', help='Input for the pipeline',  
                           default='gs://my-bucket/input')  
        parser.add_argument('--output', help='Output for the pipeline',  
                           default='gs://my-bucket/output')  
  
public interface MyOptions extends PipelineOptions {  
    @Description("My custom command line argument.")  
    @Default.String("DEFAULT") String getMyCustomOption();  
    void setMyCustomOption(String myCustomOption);  
}
```

Allows reading these options from command line as `--<option>=<value>` at submission time

PCollections

- Potentially distributed, multi-element dataset
 - Equivalent to Stream in the streaming case
 - Consumed and produced by Transforms (operators)
 - Can be created by reading from external interfaces or in memory

```
lines = p | 'ReadMyFile' >> beam.io.ReadFromText('gs://some/inputData.txt')
```

```
lines = (p | beam.Create(['To be, or not to be: that is the question: ',  
    'Whether tis nobler in the mind to suffer']))
```

```
PCollection<String> lines = p.apply("ReadMyFile",  
    TextIO.read().from("protocol://path/to/some/inputData.txt"));
```

```
static final List<String> LINES = Arrays.asList("To be, or not to be: that  
is the question: ", "Whether tis nobler in the mind to suffer ");  
p.apply(Create.of(LINES)).setCoder(StringUtf8Coder.of())
```

lines: PCollection whose elements are each a single string corresponding to a line of text in the file. In the batch case, this is a finite, bounded set of elements. In the streaming case, we could read from a continuously updating file (hot file) or from network

PCollections

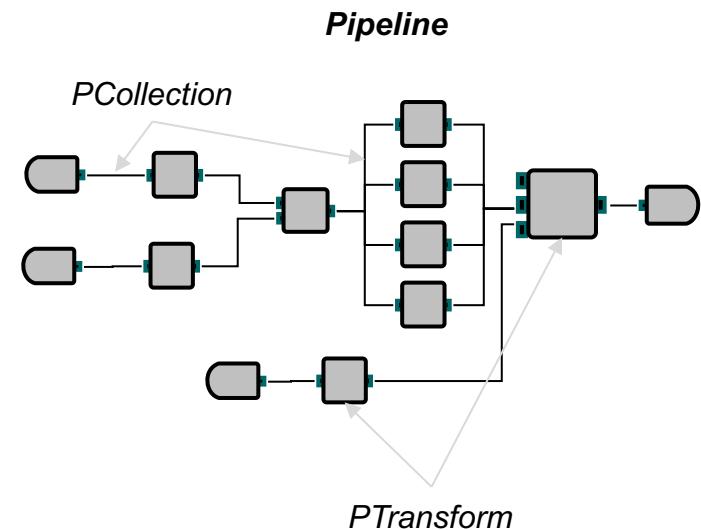
- PCollection elements can have any type
 - All elements need to have the same type
 - Need to be encoded as byte strings for distributed processing (hence need to specify encoder)
- PCollections are Immutable
 - To transform them create new Pcollections (recall RDDs)
- Random Access
 - Elements cannot be accessed in random order – collected in order of arrival (can have ordering transforms)
- Size
 - No limit on size of PCollection – can fit in memory of one machine, or distributed across multiple
 - Can be bounded (batch) or unbounded (stream)

PCollections

- Windowing
 - Used to partition unbounded Pcollections into finite sized logical sets
- Element timestamps for PCollections
 - Each element in unbounded PCollection is expected to have timestamp – assigned by source (creation or arrival time)
 - Timestamps often used for windowing
 - Batch PCollection elements are assigned the same timestamp
 - Transforms can be used to assign timestamps to elements

Transforms (Operators)

- Different operations that can be applied to 0 or more PCollections to create 0 or more PCollections
- Processing logic provided as function objects (user code)
 - Code may be run distributedly in parallel depending on the runner
- Two types of core general purpose transforms
 - ParDo and Combine – that get specialized with user code
 - Other composite transforms as well

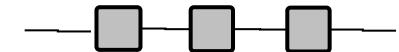


Applying Transforms

- Transforms applied to PCollection to create new PCollection
- Can be chained to apply multiple transformations

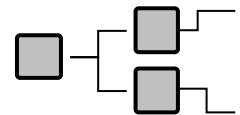
```
[Final Output PCollection] = ([Initial Input PCollection] |  
[First Transform] | [Second Transform] | [Third Transform])
```

```
[Final Output PCollection] = [Initial Input  
PCollection].apply([First Transform]).apply([Second  
Transform]).apply([Third Transform])
```



- Can build DAGs of transformations

```
[Output PCollection 1] = [Input PCollection] | [Transform 1]  
[Output PCollection 2] = [Input PCollection] | [Transform 2]
```



```
[Output PCollection 1] = [Input PCollection].apply([Transform 1])  
[Output PCollection 2] = [Input PCollection].apply([Transform 2])
```

Core Beam Transformations

- ParDo
- GroupByKey
- CoGroupByKey
- Combine
- Flatten
- Partition
- Each of these can be extended with user code
- Represent different ways of processing data
- Analogous to different functions in Spark/Spark Streaming

ParDo Transform

- Analogous to the `map` functionality in Spark/Spark Streaming and `Functor` in Streams
- Takes each element in the PCollection, applies a function (user code) to it to create output elements that are assigned to new PCollection
- Typical Applications:
 - Filter data: (select) some elements based on a condition
 - Format or type converting data (e.g. parse strings)
 - Extract parts of each element (project)
 - Apply some computation to the elements
- To apply user code within `ParDo`, we need to define a `DoFn` class
 - Needs to conform with certain guidelines

DoFn for ParDo

Need to define `process` method with custom logic. This function is passed data, element by element, from the PCollection

```
#Compute length for each word (element in PCollection)
class ComputeWordLengthFn(beam.DoFn):
    def process(self, element):
        return [len(element)]  
                                         Why this?

words = ...
word_lengths = words | beam.ParDo(ComputeWordLengthFn())

// The DoFn to perform on each element in the input PCollection.
static class WordLengthFn extends DoFn<String, Integer> {
    ...
}

PCollection<String> words = ...;
PCollection<Integer> wordLengths = words.apply(ParDo.of(new WordLengthFn()));
```

Input PCollection has elements that are strings, output PCollection has elements that are ints

DoFn for ParDo

- DoFns can be invoked in parallel (by different workers) or multiple times based on fault tolerance requirements
 - Unsafe to make the DoFn stateful in number of invocations
- Simple DoFns can be define inline using lambda functions

```
word_lengths = words | beam.ParDo(lambda word: [len(word)])
```

```
PCollection<Integer> wordLengths = words.apply( "ComputeWordLengths",
    ParDo.of(
        new DoFn<String, Integer>() {
            @ProcessElement public void processElement(@Element String word,
                OutputReceiver<Integer> out) {
                out.output(word.length()); } } ) );
```

One-to-One ParDos can be implemented as Map transforms (another special builtin)

```
word_lengths = words | beam.Map(len)
```

GroupByKey

- Applied to PCollections with key/value pairs as elements
- Like a Shuffle operation in typical Map/Shuffle/Reduce workflows
- Groups all elements in the PCollection which have the same key

cat, 1	cat, [1, 5]
dog, 5	dog, [5, 2]
and, 1	and, [1, 2]
jump, 3	jump, [3]
tree, 2	tree, [2]
cat, 5	
dog, 2	
and, 2	

- Applying to unbounded PCollections (streams) requires windowing to be performed first
- Note similarity to the Spark `groupByKey` applied to Pair RDDs

CoGroupByKey

- Like Join functionality with equality conditions on keys
- Applied to two or more PCollections with key/value pairs
- Groups all elements across two different PCollections that share the same key
 - Also performs a GroupByKey within each PCollection

```
emails = p | 'CreateEmails' >> beam.Create(emails_list)
phones = p | 'CreatePhones' >> beam.Create(phones_list)

results = ({'emails': emails, 'phones': phones} | beam.CoGroupByKey())

PCollection<KV<String, String>> emails = p.apply("CreateEmails",
Create.of(emailsList));
PCollection<KV<String, String>> phones = p.apply("CreatePhones",
Create.of(phonesList));

PCollection<KV<String, CoGbkResult>> results =
KeyedPCollectionTuple.of(emailsTag, emails) .and(phonesTag, phones)
.apply(CoGroupByKey.create());
```

CoGroupByKey

```
emails_list = [  
    ('amy', 'amy@example.com'),  
    ('carl', 'carl@example.com'),  
    ('julia', 'julia@example.com'),  
    ('carl', 'carl@email.com'),  
]  
  
phones_list = [  
    ('amy', '111-222-3333'),  
    ('james', '222-333-4444'),  
    ('amy', '333-444-5555'),  
    ('carl', '444-555-6666'),  
]
```

**CoGroup
ByKey**

```
results = [  
    ('amy', {  
        'emails': ['amy@example.com'],  
        'phones': ['111-222-3333', '333-444-  
5555']  
    }),  
    ('carl', {  
        'emails': ['carl@email.com',  
                  'carl@example.com'],  
        'phones': ['444-555-6666']  
    }),  
    ('james', {  
        'emails': [],  
        'phones': ['222-333-4444']  
    }),  
    ('julia', {  
        'emails': ['julia@example.com'],  
        'phones': []  
    }),  
]
```

Combine

- Like Aggregate functionality
- Applied to PCollection to combine elements using an appropriate aggregation function
 - Aggregation function should be commutative and associative
 - Prebuilt aggregation functions (sum, max, min) available

```
pc = [1, 10, 100, 1000]
small_sum = pc | beam.CombineGlobally(SumFn())

def bounded_sum(values, bound=500):
    return min(sum(values), bound)

small_sum = pc | beam.CombineGlobally(bounded_sum)
```

- Can define both simple and complex aggregation functions

Combine: Custom Aggregation

- Example Aggregation Function

```
class AverageFn(beam.CombineFn):  
    def create_accumulator(self):  
        return (0.0, 0)  
    def add_input(self, sum_count, input):  
        (sum, count) = sum_count  
        return sum + input, count + 1  
    def merge_accumulators(self, accumulators):  
        sums, counts = zip(*accumulators)  
        return sum(sums), sum(counts)  
    def extract_output(self, sum_count):  
        (sum, count) = sum_count  
        return sum / count if count else float('NaN')  
  
average = pc | beam.CombineGlobally(AverageFn())
```

- Recall similarity to aggregate in Pyspark
 - In the streaming setting these are applied across a window
 - Need to specify default behavior for empty windows
- Can also similarly have CombinePerKey
 - for PCollections with key,value pairs

Flatten and Partition

- Flatten
 - merges PCollections of the same type into one PCollection
 - Like a stream combiner

```
merged = ( (pcoll1, pcoll2, pcoll3) | beam.Flatten())
```

- Partition
 - splits a single PCollection into multiple PCollections based on partition condition

```
#Define partitioning function that return integer
students = ...
def partition_fn(student, num_partitions):
    return int(get_percentile(student) * num_partitions / 100)

by_decile = students | beam.Partition(partition_fn, 10)
fortieth_percentile = by_decile[4]
```

- Individual partitions can be accessed by index

Beam Writing Custom Functions

- Recall: function code gets executed distributedly
 - Can have multiple copies running in parallel
 - Can also be retried based on failure/s
 - Code must be serializable and thread safe
- Serializable
 - Should not include large amounts of data and only include serializable classes – need to ship to workers
- Thread Safe
 - Care when creating threads
- ‘Idempotent’
 - May be executed on the same data multiple times

Side Information for ParDo Transforms

- Useful to be able to have some side information
 - Not directly from the streaming data, but determined at runtime
 - E.g. Machine learning model, or parameters
- Passed as extra parameters to the process function of the ParDo

```
class FilterUsingLength(beam.DoFn):  
    def process(word, lower_bound, upper_bound=float('inf')):  
        if lower_bound <= len(word) <= upper_bound:  
            yield word  
  
small_words = words | beam.ParDo(FilterUsingLength(), 0, 3)
```

- Can similarly pass side input to other Beam transforms
 - E.g. FlatMap

Multiple Output Streams from ParDo

- Can create multiple output streams from transform

```
#Three output streams, main, above_cutoff_lengths, and marked strings
class GetWords(beam.DoFn):
    def process(self, element, cutoff_length, marker):
        if len(element) <= cutoff_length:
            yield element
        else:
            yield pvalue.TaggedOutput('above_cutoff_lengths', len(element))
        if element.startswith(marker):
            yield pvalue.TaggedOutput('marked strings', element)

results = (words | beam.ParDo(GetWords(), cutoff_length=2, marker='x')
           .with_outputs('above_cutoff_lengths', 'marked strings',
                         main='below_cutoff_strings'))
below = results.below_cutoff_strings
above = results.above_cutoff_lengths
marked = results['marked strings']
```

- Access streams by tags or by keys

Pipeline I/O – Reading and Writing

- Built in connectors
 - Memory, File, Cloud Storage, Databases, Pub-sub...

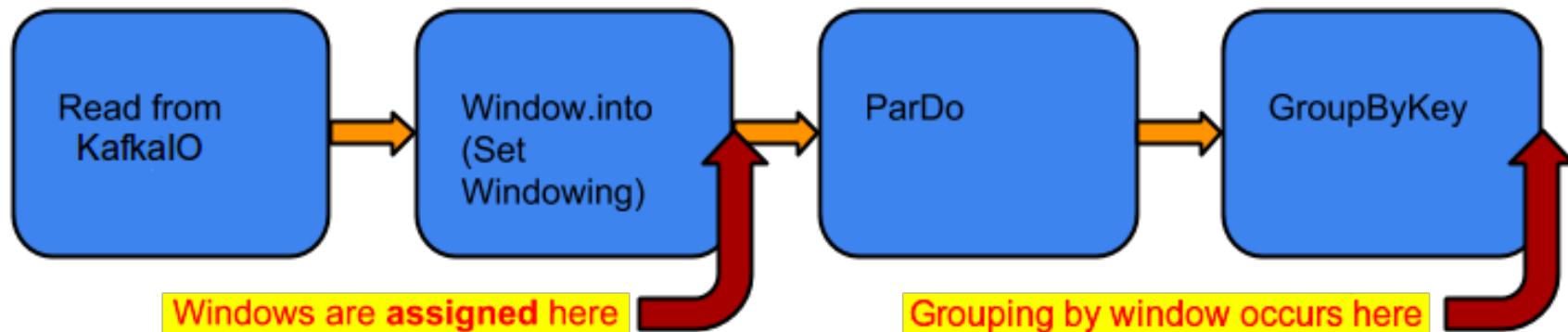
```
output | beam.io.WriteToText('gs://some/outputData')
lines = p | 'ReadFromText' >> beam.io.ReadFromText('input-* .csv')
```

```
output.apply(TextIO.write().to("gs://some/outputData"));
p.apply("ReadFromText",
TextIO.read().from("protocol://my_bucket/path/to/input-* .csv");
```

	File	Messaging	Database
Java	FileIO, AvroIO, TextIO, TFRecordIO, XmlIO, TikalIO, ParquetIO	Kinesis, Kafka, Pub/Sub, MQTT, JMS	Cassandra, Hadoop InputFormat, Hbase, Hive, Apache Kudu, Apache Solr, Elasticsearch, Google BigQuery, Google Cloud Bigtable, Google Cloud Datastore, Google Cloud Spanner, JDBC, MongoDB, Redis
Python	avroio, textio, tfrecordio, vcfio	GC Pub/Sub	Google BigQuery, Cloud Datastore

Beam Windowing

- Windowing used to partition PCollections based on individual tuple timestamps
 - Support attribute delta (specifically based on timestamp/arrival time)
- Each element in a PCollection is assigned to one or more windows
- Each individual window contains a finite number of elements
- All transforms and aggregations are applied on a window by window basis
 - E.g. GroupByKey groups elements by key and window
- Windows actually used when aggregation needs to be performed



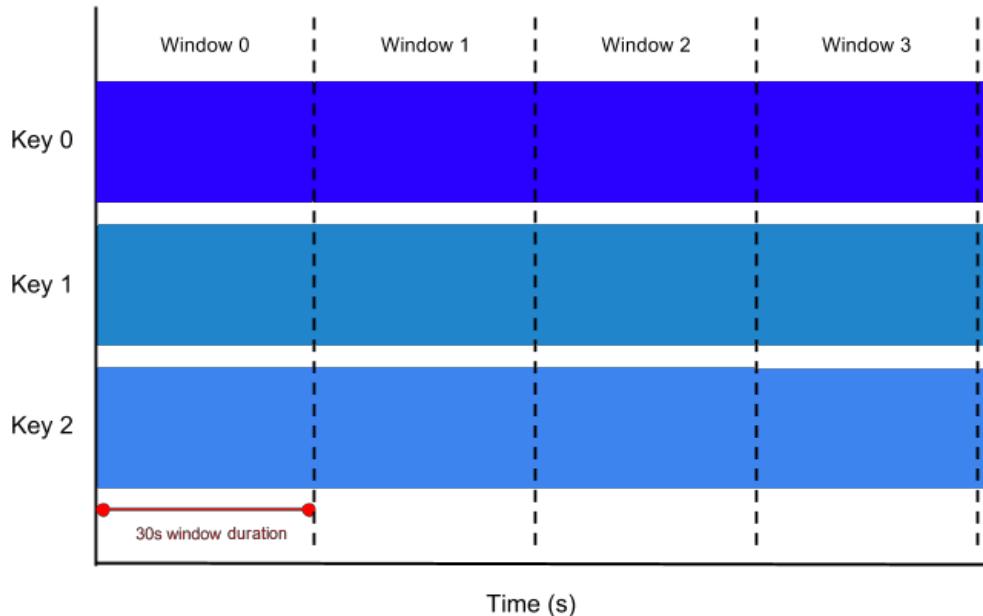
From <https://beam.apache.org/documentation/programming-guide/>

Built-in Windowing Support

- Delta-based Windows
 - Fixed Time Windows (Tumbling Windows)
 - Sliding Time Windows
- Per-Session Windows
- Single Global Window
- Calendar-based Windows
- All windowing performed per Key
- Custom Windows can also be defined with a `WindowFn`

From <https://beam.apache.org/documentation/programming-guide/>

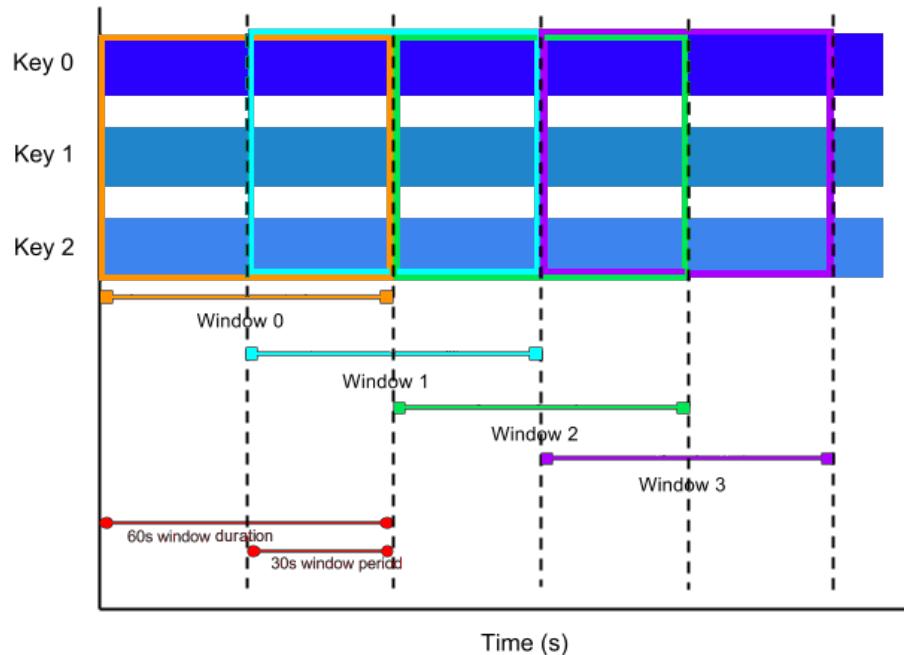
Beam Fixed Time Windows



```
PCollection<String> items = ...;  
PCollection<String> fixedWindowedItems = items.apply(Window.<String>into  
    (FixedWindows.of(Duration.standardSeconds(30))));
```

```
from apache_beam import window  
fixed_windowed_items=(items | 'window' >> beam.WindowInto(window.FixedWindows(30)))
```

Beam Sliding Time Windows



```
from apache_beam import window
sliding_windowed_items=( items | 'window' >>
                           beam.WindowInto(window.SlidingWindows(60, 30)))

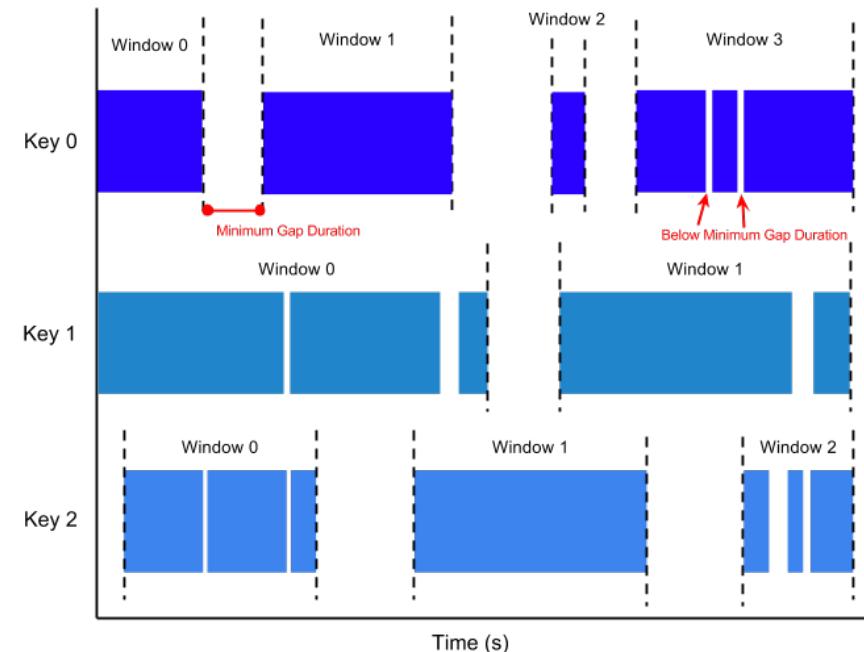
PCollection<String> items = ...;
PCollection<String> slidingWindowedItems = items.apply(
    Window.<String>into(SlidingWindows.of(Duration.standardSeconds(60)).
        every(Duration.standardSeconds(30))));
```

Beam Session Windows

- Defined for irregularly spaced data
 - where gap in data arrival specifies window boundaries

```
from apache_beam import window
session_windowed_items = ( items | 'window'
    >> beam.WindowInto(window.
        Sessions(10 * 60)))

PCollection<String> items = ...;
PCollection<String> sessionWindowedItems =
    items.apply(Window.<String>
        into(Sessions.withGapDuration
            (Duration.standardMinutes(10))));
```



- New window started when the gap in arrival is greater than the specified minimum value

Watermarks and Late Data

- Used to account for difference between event time and arrival time
 - Can specify max lateness in data arrival to allow it to be added to window
 - E.g. if max lateness is 30 seconds and we have a 5:00 minute window, then Beam will wait 30 seconds for such data
 - If at 5:29 in real time, we have data with event time 4:45 then it will be added to window 1
 - If at 5:34 in real time we have data with event time 4:43 – this data will be discarded – since it arrived later than the max lateness
- Only supported in Java, not in Python
- Watermark estimated by Beam, based on data arrival patterns, and user specification

```
PCollection<String> items = ...;  
PCollection<String> fixedWindowedItems = items.apply(  
    Window.<String>into(FixedWindows.of(Duration.standardMinutes(1)))  
    .withAllowedLateness(Duration.standardDays(2)));
```

Timestamps and PCollections

- Streaming data comes with timestamps naturally
 - Batch data does not
- Need to tell Beam how to extract/assign timestamps from the data
- Apply ParDo transforms to do this

```
class AddTimestampDoFn(beam.DoFn):  
    def process(self, element):  
        unix_timestamp = extract_timestamp_from_log_entry(element)  
        yield beam.window.TimestampedValue(element, unix_timestamp)  
  
timestamped_items = items | 'timestamp' >> beam.ParDo(AddTimestampDoFn())  
  
PCollection<LogEntry> unstampedLogs = ...;  
PCollection<LogEntry> stampedLogs = unstampedLogs.apply(ParDo.of(new DoFn<LogEntry,  
LogEntry>(){  
    public void processElement(@Element LogEntry el, OutputReceiver<LogEntry> out){  
        Instant logTimeStamp = extractTimeStampFromLogEntry(element);  
        out.outputWithTimestamp(element, logTimeStamp);  
    }});
```

Window Triggers

- Determine when results from the window are emitted
- Event time triggers
- Processing time triggers
- Data-driven triggers
- Composite triggers

Window Triggers and Composition

- Event Triggers: operate on the event time (data time), e.g.
AfterWaterMark trigger
 - Processing performed when watermark passes end of window
 - Recall watermarks and Structured Spark Streaming
- Processing Time Triggers: operate on wall clock time, e.g.
AfterProcessingTime trigger
 - Processing performed interval after the first element in window is received
 - Supports count and time based interval specification
- Data-driven Triggers: element count, e.g. AfterCount trigger
 - Processing performed after receiving a certain number of tuples
- Different types of triggers can be composed

```
AfterWatermark(early=AfterProcessingTime(delay=1*60), late=AfterCount(1))
```

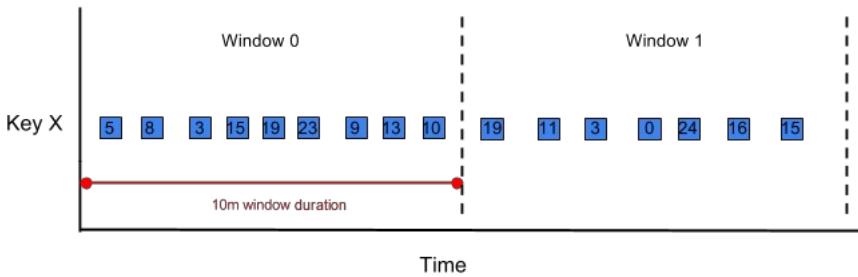
```
AfterWatermark.pastEndOfWindow()  
    .withEarlyFirings( AfterProcessingTime .pastFirstElementInPane()  
        .plusDuration(Duration.standardMinutes(1))  
    .withLateFirings(AfterPane.elementCountAtLeast(1))
```

Setting Window Triggers

- Associated with the `WindowInto` transform

```
pcollection | WindowInto( FixedWindows.of(1 * 60),  
                         trigger=AfterProcessingTime.of(10 * 60),  
                         accumulation_mode=AccumulationMode.DISCARDING)  
  
pc.apply(Window..into(FixedWindows.of(1, TimeUnit.MILLISECONDS))  
        .triggering(AfterProcessingTime.pastFirstElementInPane())  
        .plusDelayOf(Duration.standardMinutes(1))) .discardingFiredPanes();
```

- Accumulation Mode determines what happens to windows on each firing (since repeated firings possible)



accumulatingFiredPanes () Mode
First trigger firing: [5, 8, 3]
Second trigger firing: [5, 8, 3, 15, 19, 23]
Third trigger firing: [5, 8, 3, 15, 19, 23, 9, 13, 10]

discardingFiredPanes () Mode
First trigger firing: [5, 8, 3]
Second trigger firing: [15, 19, 23]
Third trigger firing: [9, 13, 10]

Handling Late Data

- In Python – use parameter `allowed_lateness`

```
pc = [Initial PCollection] pc |  
beam.WindowInto( FixedWindows(60),  
trigger=trigger_fn,  
accumulation_mode=accumulation_mode,  
allowed_lateness=Duration(seconds=2*24*60*60))  
# 2 days
```

- In Java – use `withAllowedLateness()` to specify

References

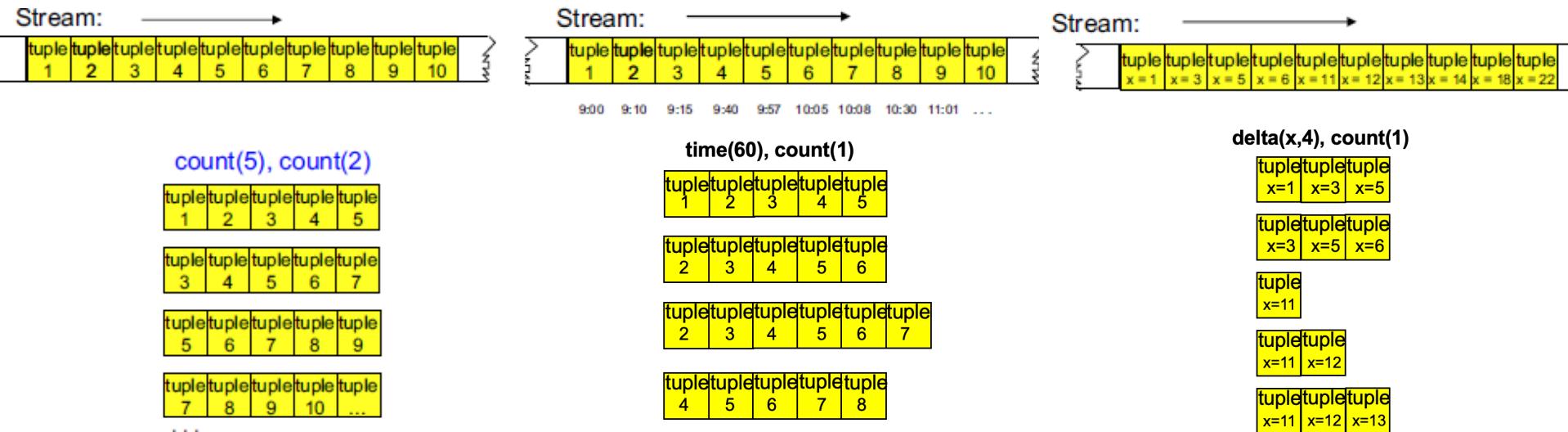
- Apache Beam Documentation:
<https://beam.apache.org/documentation/programming-guide/>
- Apache Beam Community:
<https://beam.apache.org/community/contact-us/>

Relational Stream Processing Streaming Optimizations

Objectives

- Quick Takeaways from Lecture 3
 - Windowing
 - Apache Beam
- Relational Streaming
 - Operator Concepts
- Streaming Optimizations

Windowing



Window Properties

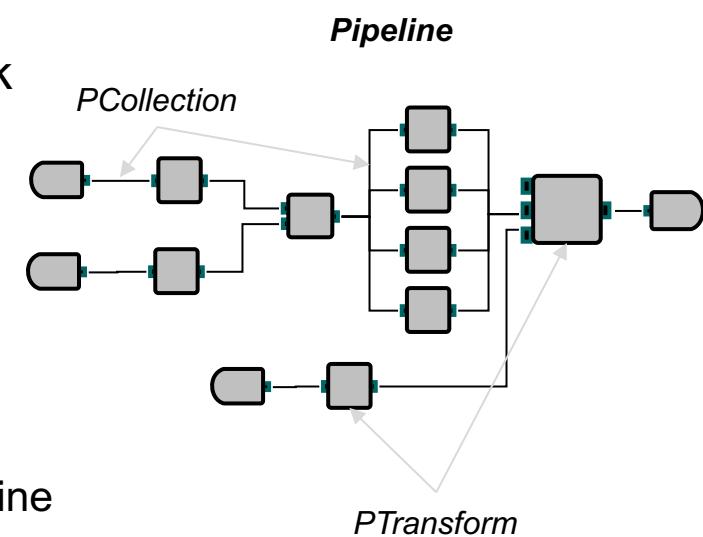
- Three properties that define a window
 - Eviction policy
 - Defines how large a window can get
 - Determines which older tuples are removed from the window
 - Trigger policy
 - When an operation, such as aggregation, takes place as new tuples arrive into the window
 - Partitioning
 - Maintains separate windows for each group of tuples with the same grouping key value
- Window types
 - Tumbling windows:
 - Non-overlapping sets of consecutive tuples
 - Defined only with an eviction policy
 - Sliding windows:
 - Windows formed by adding new tuples to the end of the window and evicting old tuples from the beginning of the window
 - Defined with both an eviction and a trigger policy

Apache Beam

- Open source unified model
 - Composition language for data pipelines
 - For batch and streaming jobs
- Multiple language SDK
 - Java, Python, Go, Scala
- Multiple execution frameworks (runners)
 - Apache Apex 
 - Apache Flink 
 - Apache Gearpump 
 - Apache Samza 
 - Apache Spark 
 - Google Dataflow 
 - IBM Streams

Apache Beam Programming

- Define driver program using one of Beam SDK languages
 - Java, Python, Go
- Core abstractions
 - *Pipeline* (application flowgraph)
 - Encapsulates entire data processing task
 - Needs to specify where and how to run
 - *PCollection* (Dataset or Stream)
 - Bounded datasets for batch processing
 - Unbounded for stream processing
 - *PTransform* (Operator)
 - Data processing operation or step in the pipeline
 - Multiple special I/O transforms for PCollections



Apache Beam Programming

```
def mypipelinerun():
    p = beam.Pipeline(options=PipelineOptions())

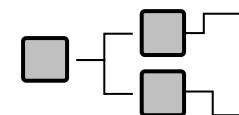
lines = p | 'ReadMyFile' >> beam.io.ReadFromText('gs://some/inputData.txt')

words = (lines | 'SplitLines' >> lines.FlatMap(str.split)

#Compute length for each word (element in PCollection)
class ComputeWordLengthFn(beam.DoFn):
    def process(self, element):
        return [len(element)]

word_lengths = words | beam.ParDo(ComputeWordLengthFn())

[Output PCollection 1] = [Input PCollection] | [Transform 1]
[Output PCollection 2] = [Input PCollection] | [Transform 2]
```



Core Beam Transformations

- ParDo
 - Map, FlatMap
- GroupByKey
- CoGroupByKey
- Combine
- Flatten
- Partition
- Each of these can be extended with user code
- Represent different ways of processing data

Beam Window Triggers

- Determine when results from the window are emitted
- Event time (data time) triggers
- Processing time (wall clock time) triggers
- Data-driven triggers
- Composite triggers

Window Triggers and Composition

- Event Triggers: operate on the event time (data time), e.g.
AfterWaterMark trigger
 - Processing performed when watermark passes end of window
 - Recall watermarks and Structured Spark Streaming
- Processing Time Triggers: operate on wall clock time, e.g.
AfterProcessingTime trigger
 - Processing performed interval after the first element in window is received
 - Supports count and time based interval specification
- Data-driven Triggers: element count, e.g. AfterCount trigger
 - Processing performed after receiving a certain number of tuples
- Different types of triggers can be composed

```
pcollection | WindowInto( FixedWindows(1 * 60),  
trigger=AfterWatermark(late=AfterProcessingTime(10 * 60)),  
allowed_lateness=10, accumulation_mode=AccumulationMode.DISCARDING)
```

References

- Apache Beam Documentation:
<https://beam.apache.org/documentation/programming-guide/>
- Apache Beam Community:
<https://beam.apache.org/community/contact-us/>

Operator Concepts

Operator Concepts

- Operators are stream manipulators
- Operators provide a declarative interface
- Some important operator properties
 - Operator state
 - Selectivity and arity
 - Parameters
 - Output assignments and output functions
 - Windowing
 - Punctuations

Operator State

- Three classes of operators categories
 - Stateless
 - Stateful
 - Partitioned stateful

Stateless Operators

- Do not maintain internal state across tuples
- Processes on a tuple-by-tuple basis
- E.g. Projection with Functor, Map, ParDo
- Advantages:
 - Can be parallelized easily
 - No need for synchronization in a multi-threaded context
 - Can be restarted upon failures

Example Stateless Operators

```
1 stream<rstring greeting, uint32 id> Message = Functor(Beat) {
2     output
3         Message: greeting = "Hello World! (" + (rstring) Beat.id + ")";
4 }
```

```
lines = sc.parallelize([1, 4, 3])
//read from file
lines2 = lines.map(lambda x: x*x)
res = lines2.collect()
for num in res:
    print num
```

```
class ComputeWordLengthFn(beam.DoFn):
    def process(self, element):
        return [len(element)]

words = ...
word_lengths = words |
beam.ParDo(ComputeWordLengthFn())
```

Stateful Operators

- Create and maintain state as tuples processed
- Such state, along with internal algorithm, affects the results
- e.g. DeDuplicate or Join operators or reduceByKeyAndWindow
 - *tuple is considered a duplicate if it shares the same key with a previously seen tuple within a pre-defined period of time*
- Runtime support is challenging
 - Require synchronization in a multi-threaded context
 - No direct data parallelization
 - Require some persistence mechanism for fault-tolerance

Partitioned Operators

- Keep state across tuples
- State can be divided into *partitions*
- Each partition depends on independent segments of the data stream
- Another way of looking at it
 - Input stream is divided into non-overlapping *sub-streams*
 - Sub-streams are processed independently
 - Each sub-stream has its own state
- Example: Computing Volume Weighted Average Price (VWAP) on all stock tickers
 - each transaction contains an attribute *ticker* (e.g., “IBM”, “AAPL”, “GOOG”), a *volume*, and a *price*
 - an application wants to compute a VWAP for each ticker over the last 10 transactions on that particular ticker
- `partitionBy` keyword

Example Partitioned Operators

```
//Create a streaming reduce with a 1 second batch, 4 second window
ipDstream = accessLogs.map(lambda x:(x.getIpAddress(),1))
ipCountDstream = ipDstream.reduceByKeyAndWindow(
    lambda (x,y): x+y,
    lambda (x,y): x-y,
    Seconds(4),
    Seconds(1))
```

Not explicitly partitioned...

Recall partitioning strategy

```
Data = sc.parallelize([("a",1),("b",2),("c",3)])
Res2 = Data.partitionBy(new HashPartitioner(100)).persist()
```

Default mode for Apache Beam is partitioned by key

Operator Selectivity and Arity

- Selectivity: captures the relationship between the number of tuples produced and consumed by an operator
 - Fixed selectivity
 - Variable selectivity
- Fixed: Consumes a fixed number of tuples to generate a fixed number of output tuples
 - E.g. The Map operator with one input tuple to generate one output tuple
- Variable: exhibits a non-fixed relationship between the number of tuples consumed and produced
 - E.g. The Join operator may generate 0, 1, or more output tuples, depending on the match condition, window contents

Operator Arity

- Arity refers to the number of ports an operator has
- An important class is single input / single output operators
- Selectivity of single input / single output operators
 - One-to-one (1:1) – applies a simple map operation
 - One-to-at-most-one (1:[0,1]) – applies a simple filter operation
 - One-to-many (1: N) – performs expansion
 - Many-to-one (M :1) – performs data reduction
 - Many-to-many (M : N) – the remaining case

Punctuation

- Special markers inside stream
 - Window punctuations
 - Final punctuations
- Window: marks boundaries so that groups of tuples can be treated together
 - E.g.: Aggregate results from a Join operator
 - Operators that generate output tuples in batches typically insert a window punctuation after each batch
 - Punctuation based windows can be used to operate on these batches

Windows

- Sorting, aggregating, or joining data in a relational table
 - All data in the table can be processed
- For streams, data flows continuously
 - No beginning, no end
 - Requires a different paradigm for sorting, aggregating and joining
 - Can only work with a subset of consecutive tuples
- This finite set of tuples is called a window

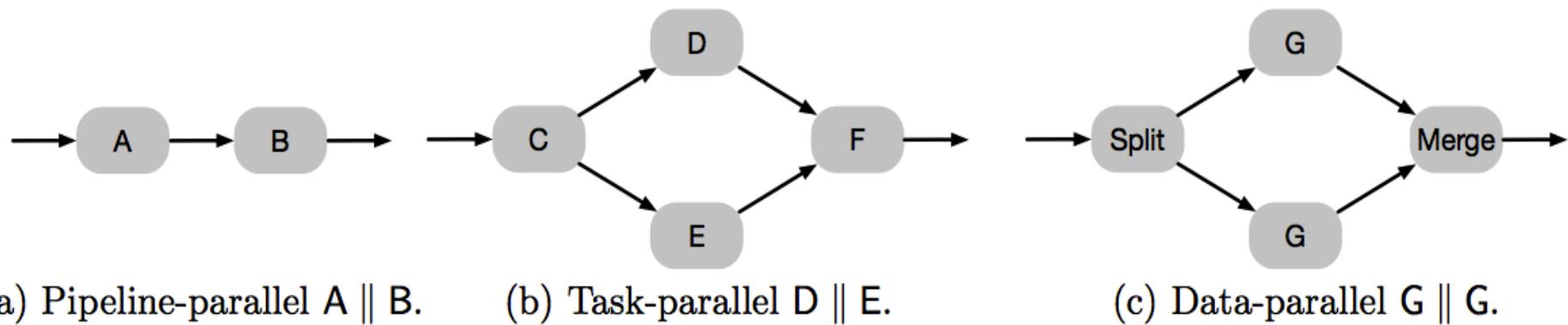
Streaming Optimizations

- Stream processing
 - Data flow graphs
- Appears in many areas of computer science
 - Databases
 - Systems (OS/Distributed)
 - Programming Languages
- These are different areas
 - Same underlying optimizations
 - Different terminology (operator vs box, hoisting vs. push-down)
 - Different assumptions (cyclic vs acyclic graphs, shared memory vs distributed)
- Of interest
 - Safety discussions (depends on assumptions, app)
 - Profitability discussions (depends on assumptions, app, workload, resources)

Kinds of Optimization

- Three dimensions
 - Does it change the graph?
 - Optimizations that involve graph transformations change the graph.
E.g.: Fission
 - Some optimizations do not require transformation. E.g. Load balancing.
 - Does it impact the application semantics?
 - Ideally, an optimization should not impact the application semantics.
 - There are a few exceptions: load shedding, algorithm selection
 - Is it dynamic?
 - If the optimization can be applied at runtime, potentially based on current resource and workload availability, then it is dynamic
 - Static optimizations are often done at compile-time

Types of Parallelism



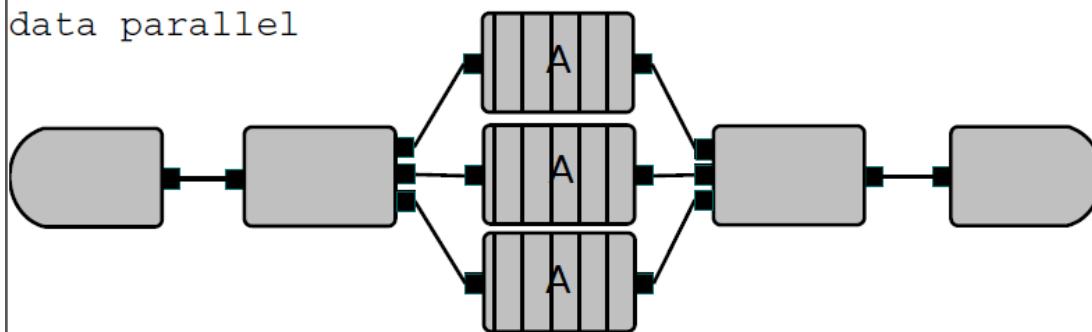
- (a-pipeline) Successive data items processed by successive tasks at the same time
- (b-task) Same data items processed by multiple tasks at the same time
- (c-data) Successive data items processed by different instances of the same task at the same time

Types of Parallelism

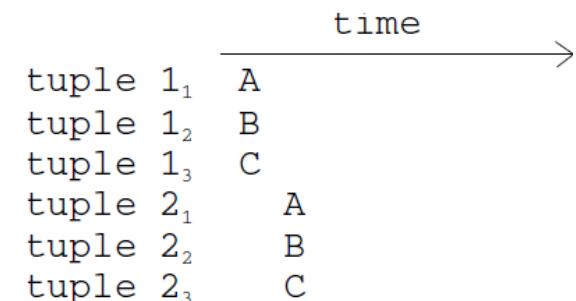
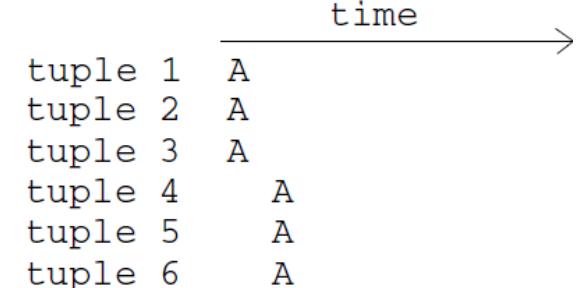
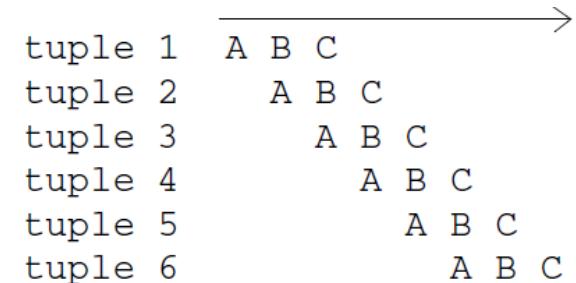
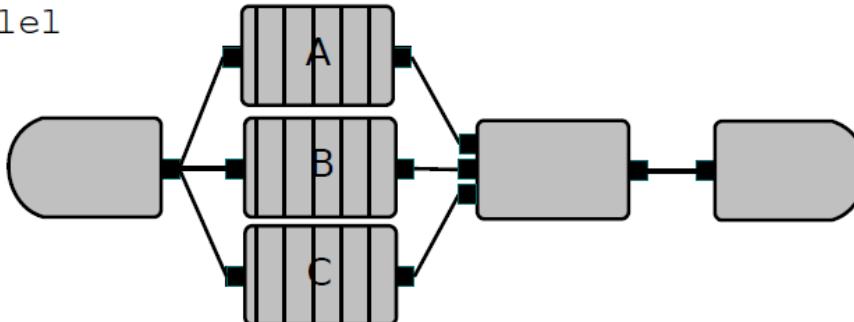
pipelined parallel



data parallel



task parallel

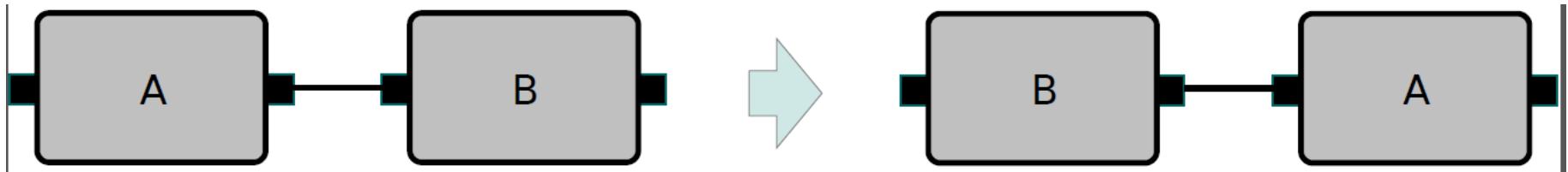


Optimization Catalog

Optimization	Graph	Semantics	Dynamics
Operator re-ordering	changed	unchanged	(depends)
Redundancy elimination	changed	unchanged	(depends)
Operator separation	changed	unchanged	static
Fusion	changed	unchanged	(depends)
Fission	changed	(depends)	(depends)
Placement	unchanged	unchanged	(depends)
Load balancing	unchanged	unchaged	(depends)
State sharing	unchanged	unchanged	static
Batching	unchanged	unchanged	(depends)
Algorithm selection	unchanged	(depends)	(depends)
Load shedding	unchanged	changed	dynamic

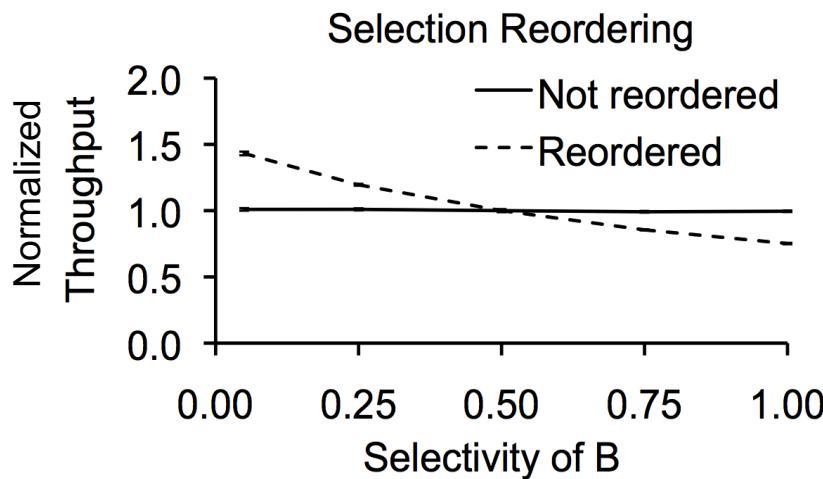
Operator Reordering

- Also known as
 - Hoisting, sinking, rotation, push-down
- *Move more selective operators upstream to filter data early.*



Profitability of Operator Reordering

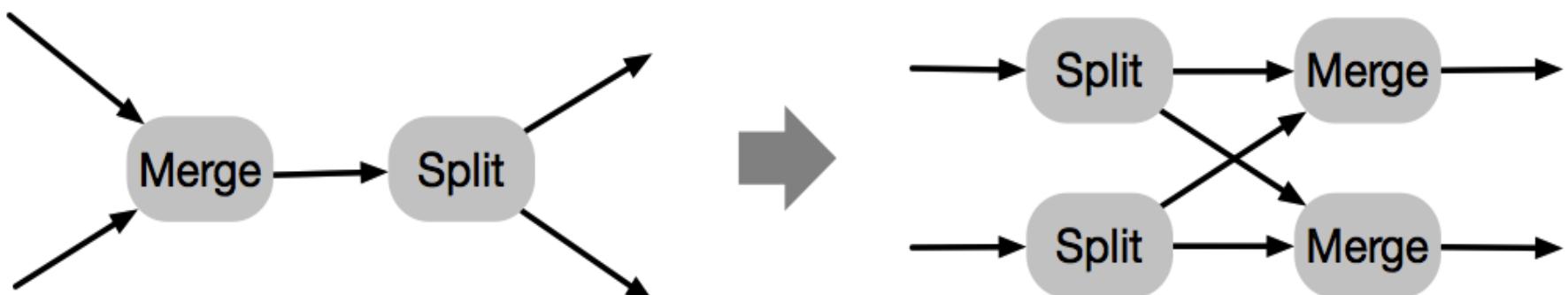
- $c(A)$: compute cost of A (per unit tuple)
- $s(A)$: selectivity of A
- Compute cost (per input tuple) without reordering
 - $c(A) + s(A)c(B)$
- Compute cost with reordering
 - $c(B) + s(B)c(A)$
- Assume, $c(A)=c(B)=1$ and $s(A)=0.5$ Given a fixed compute budget



Safety of Operator Reordering

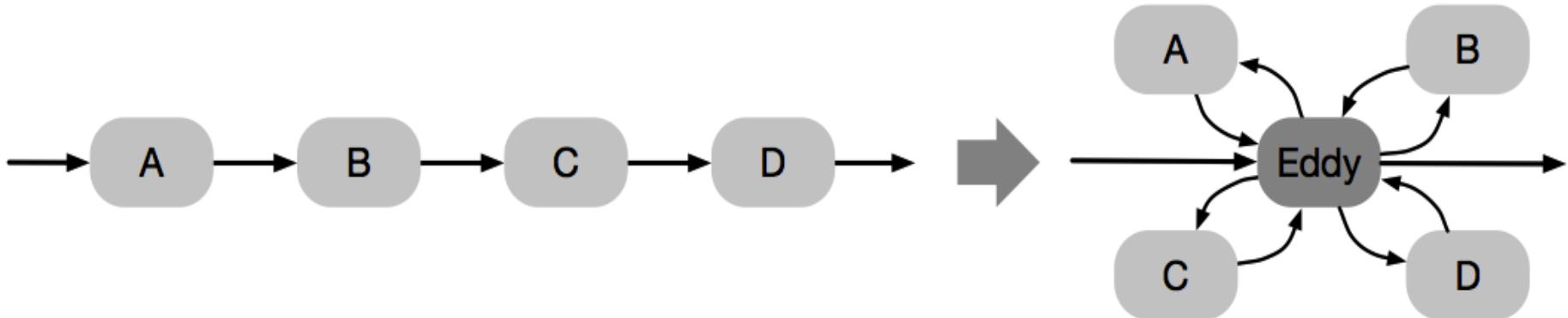
- Ensure attribute availability
 - The second operator should only rely on the attributes of the data that are already available before the first operator
 - i.e., the set of attributes that B reads from a data item must be disjoint from the set of attributes that A writes to a data item.
- Ensure commutativity
 - The order in which the operations are applied does not matter
 - E.g. filtering and projection
 - In the presence of attribute availability, a sufficient condition for commutativity is if A and B are stateless

Variant: Split-Merge Rotation



- Move split before the merge
- This is often called a *shuffle*
- The reordered version is more scalable
 - You can increase the number of merged/splitted streams
- Used in M/R systems as well

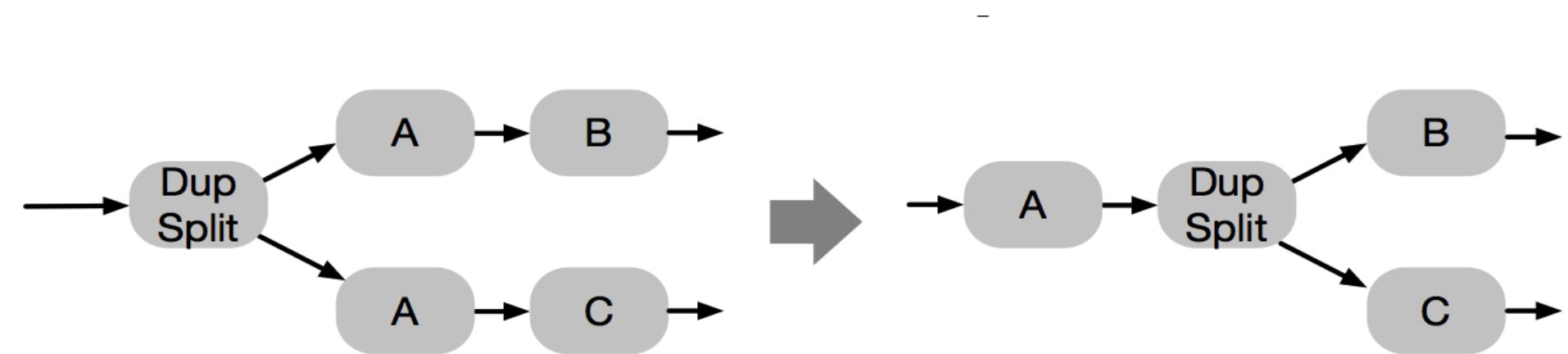
Discussion: Dynamic Reordering



- The Eddy operator
 - Dynamically route the tuples
 - Has to keep track of progress
 - Can determine the order based on current selectivities
 - Selectivity may not be known at compile-time or can change at runtime
 - A weakness: Assumes selectivities are independent of the order

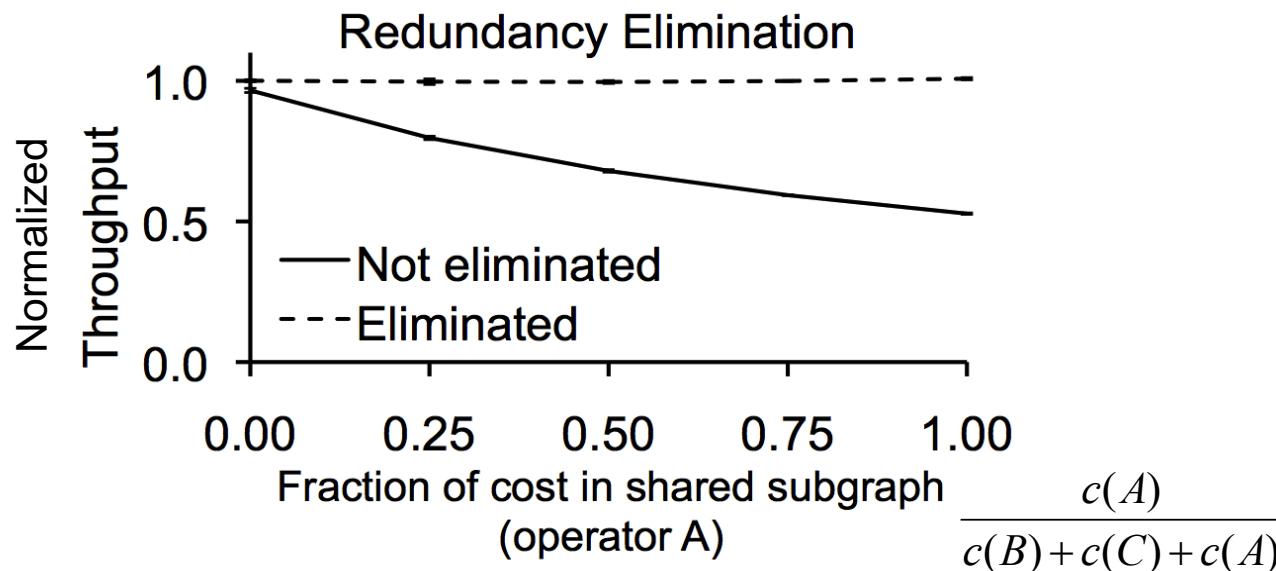
Redundancy Elimination

- Also known as
 - Subgraph sharing, multi-query optimization
- *Eliminate redundant computations.*



Profitability of Redundancy Elimination

- In general, removing redundancy does not hurt performance
- It improves performance when
 - The shared subgraph does significant work
- Before elimination: $c(B) + c(C) + c(A) + c(A)$
- After elimination: $c(B) + c(C) + c(A)$



Safety of Redundancy Elimination

- Ensure same algorithm
 - The redundant operators must perform the same operation
 - E.g. Same code, or algebraic equality
- Ensure combinable state
 - Assume the operator counts the number of tuples and we have a content based split that uses a partition by attribute
 - The result before and after redundancy elimination will be different, even though the operators do the same thing
 - If the shared operator is stateless, then no issue.
 - If stateful, then the state maintained over the original stream must be independent for individual sub-streams.

Dynamic Redundancy Elimination

- Static redundancy elimination
 - No-ops, dead-subgraphs
 - Shared subgraphs
 - if applications are all available
- Dynamic redundancy elimination
 - Needed when multi-tenancy is present
 - Applications can be submitted at different times
 - Multi-query optimization at run-time (submit/cancel)

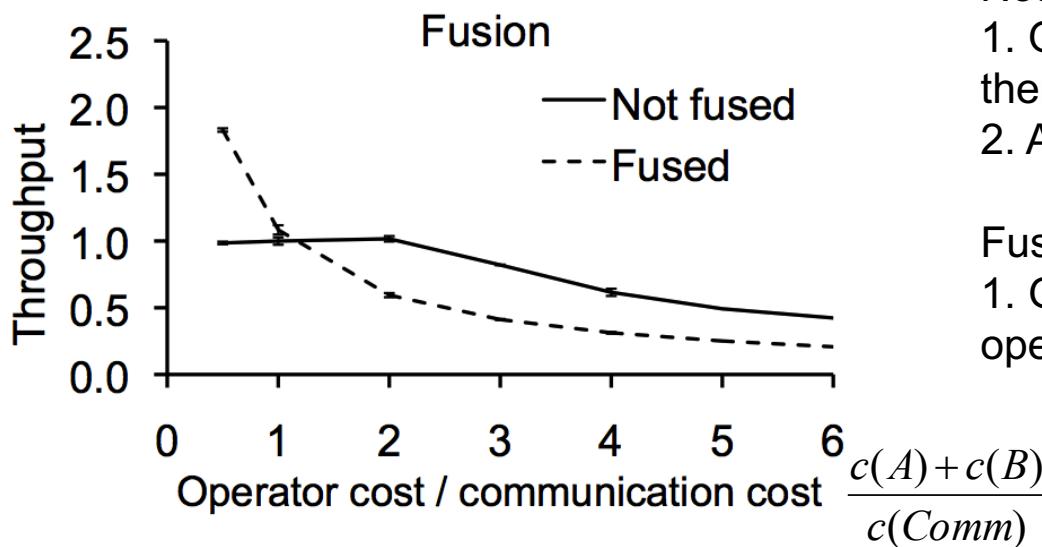
Fusion

- Also known as
 - Superbox scheduling
- *Avoid the overhead of data serialization and transport.*



Profitability of Fusion

- Fusion trades communication cost against pipeline parallelism
- If the cost of one of the operators is small (relative to communication cost)
 - Better if we fuse them (avoid the overheads)



Not fused setting:

1. Operator cost < communication cost, then bounded by communication cost.
2. After that goes down with operator cost

Fused setting:

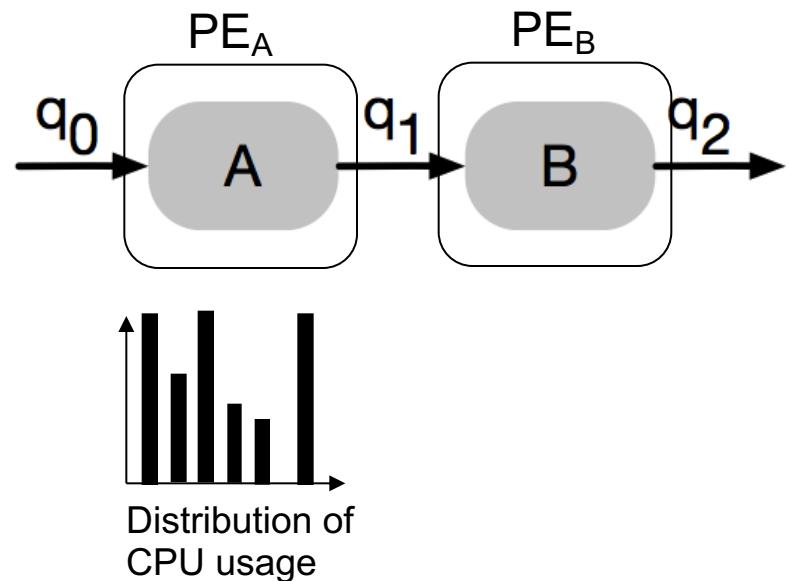
1. Goes down with operator cost (but fused operator has twice the cost of unfused)

Illustration: Fusion in the SPL Language

- Compiler can fuse in different modes
- **Default Fusion**
 - `sc -p FDEF -M <Namespace>::<MainCompositeName>`
- **Optimized Fusion**
 - `sc -p FOPT -M <Namespace>::<MainCompositeName>`
- **Need to run in profile mode**
 - `sc -P <Nsamples> -S <samplingrate> -p FOPT -M <Namespace>::<MainCompositeName>`
 - **Store** `Nsamples` from which profiles are computed using reservoir sampling
 - **Sample every** `samplingrate` (≤ 1) tuples – affects performance
 - Submit and cancel job to compute profiles

Profile Mode

- Run each operator inside its own PE
- Collect and compute statistics
 - Compute resources used
 - Memory, CPU, disk
 - Communication costs
- Determine optimum fusion assuming certain available compute from each core
 - Multi-objective optimization
 - Re-optimize
- Notes:
 - Measuring statistics has performance overheads
 - Need dedicated cores when profiling



Estimated from $N_{samples}$ samples, each of which is measured every $1/\text{samplingrate}$ tuples

Operator → PE: config placement Clause

- Partitioning allows you to specify operators that are to be fused together or kept apart
 - Uses config placement clause
- partitionColocation
 - Operator instances with the same partitionColocation value must run in the same partition
- partitionIsolation
 - Operator instance has a partition by itself
 - No other operator instances can run in that partition
- partitionExlocation
 - Operator instances with the same partitionExlocation value must run in different partitions

Operator → PE: Partitions

```
Stream <> S1 = MyOp1 { ...
    config placement: partitionColocation("fuseMe");
}
Stream <> S2 = MyOp2 { ...
    config placement: partitionColocation("fuseMe"),
        partitionExlocation("dontFuse");
}
Stream <> S3 = MyOp3 { ...
    config placement: partitionIsolation;
}
Stream <> S4 = MyOp4 { ...
    config placement: partitionExlocation("dontFuse")
}
```

Safety of Fusion

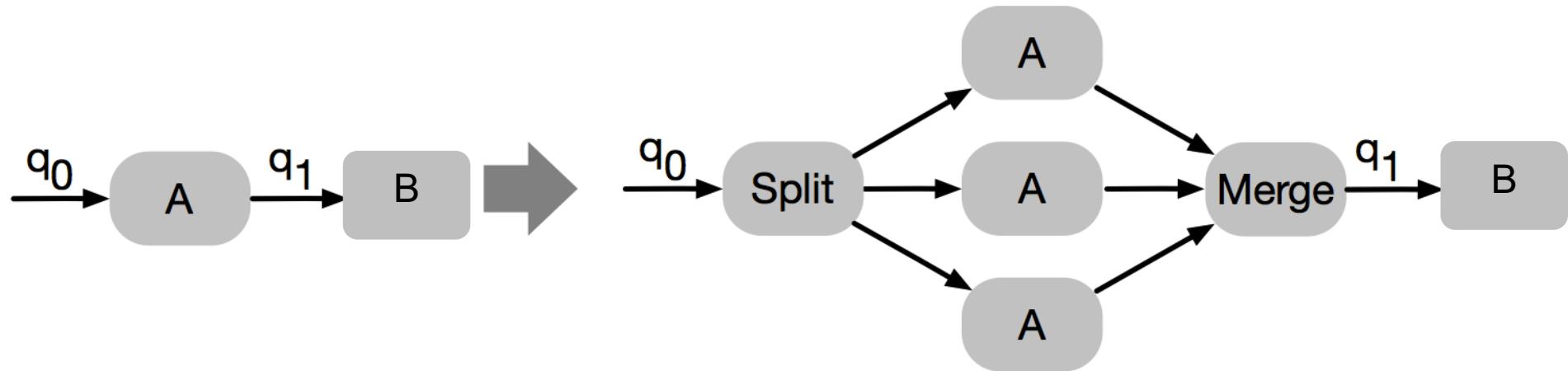
- Ensure resource kinds
 - All resources needed by A and B are available on a single process/host
- Ensure resource amounts
 - Total amount of resources required are available on a single process/host
- Avoid infinite recursion
 - Cycles in the stream graph can result in stack overflow

Dynamic Fusion

- Typically done at compile-time
 - Fusion across hosts at runtime requires heavy machinery
- Dynamic fusion using multiple threads in the same process
 - See “Auto-Pipelining for Data Stream Processing”
 - Dynamic profiling
 - Automatic thread/buffer insertion

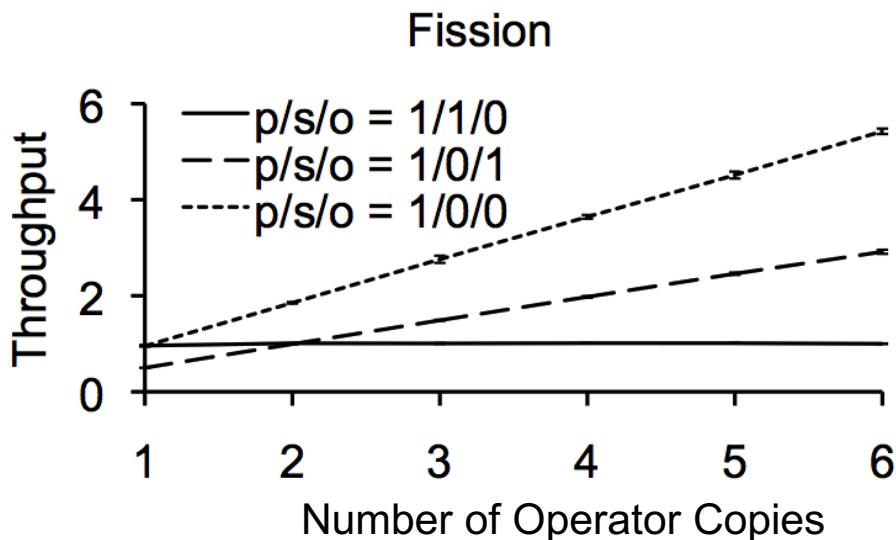
Fission

- Also known as
 - Partitioning, data parallelism, replication
- *Parallelize computations.*



Profitability of Fission

- Fission is profitable if the replicated operator is costly enough to be the bottleneck of the system
- Splits and merges can introduce overhead
- p/s/o: cost of the parallel part (A) / cost of the sequential part (B) / overhead



p/s/o (1/1/0): No matter how much we speed up parallel part, sequential part will be bottleneck
p/s/o (1/0/0): Speeds up linearly with number of copies
p/s/o (1/0/1): Until overhead is met (i.e. 2) worse than no parallel

Safety of Fission

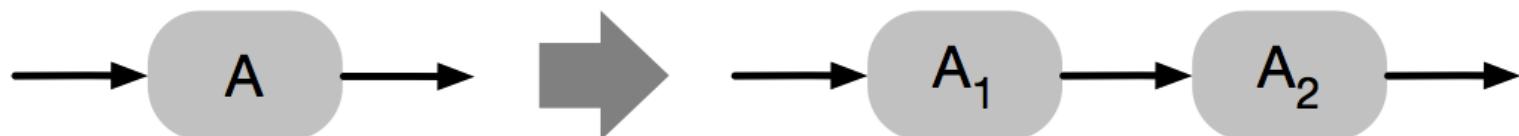
- If there is state, it needs to be partitioned stateful
 - Need to use a splitting strategy that assigns tuples to parallel channels based on the partition by attribute
- If ordering is required, merge in order
 - Need to use sequence numbers to re-order tuples at the merge
 - If there are selective operators, blockage can happen
 - Use pulses to address this

Dynamic Fission

- Dynamism is possible
 - Can increase/decrease the number of parallel channels based on the workload availability
- A number of challenges exist
 - Detecting bottlenecks
 - Congestion and throughput
 - Local vs remote congestion
 - Locating a good operating point
 - Providing SASO properties: Settling time / Accuracy / Stability / Overshoot
 - Migration of state for partitioned stateful operators
 - Consistent hashing

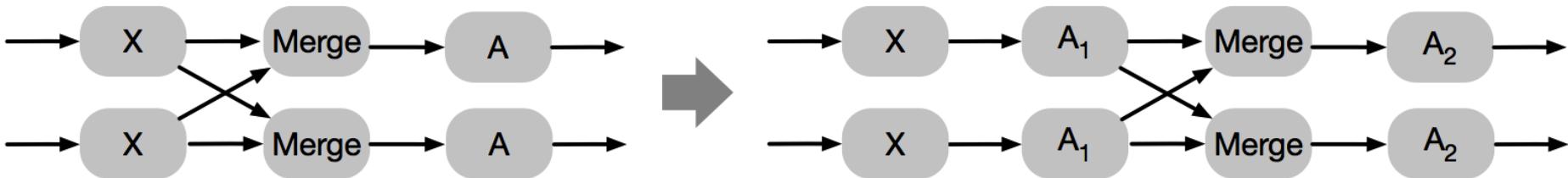
Operator Separation and Pipelines

- Also known as
 - Decoupled Software Pipelining
- *Separate operators into smaller computational steps (inverse of Fusion)*



Profitability of Operator Separation

- This is an enabler optimization
 - Operator reordering often benefits from separation
 - Fission benefits from separation
- Example: Combiners in M/R style shuffles

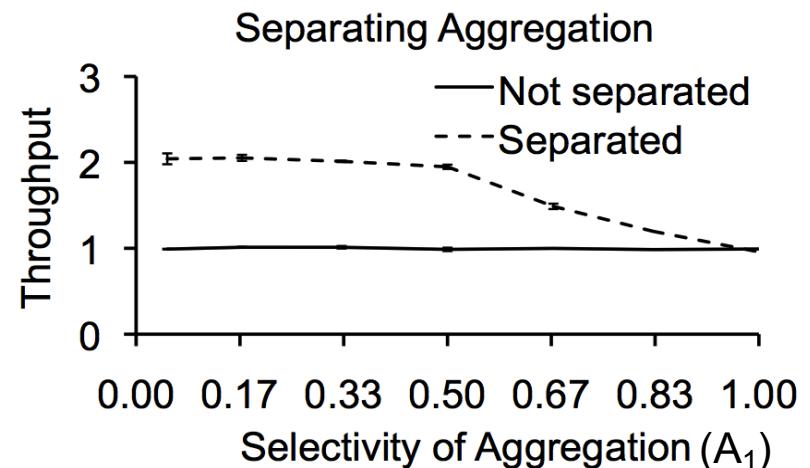


Assume $c(\text{Merge}) = 0.5$, $c(A) = 0.5$, $c(X) \sim 0$

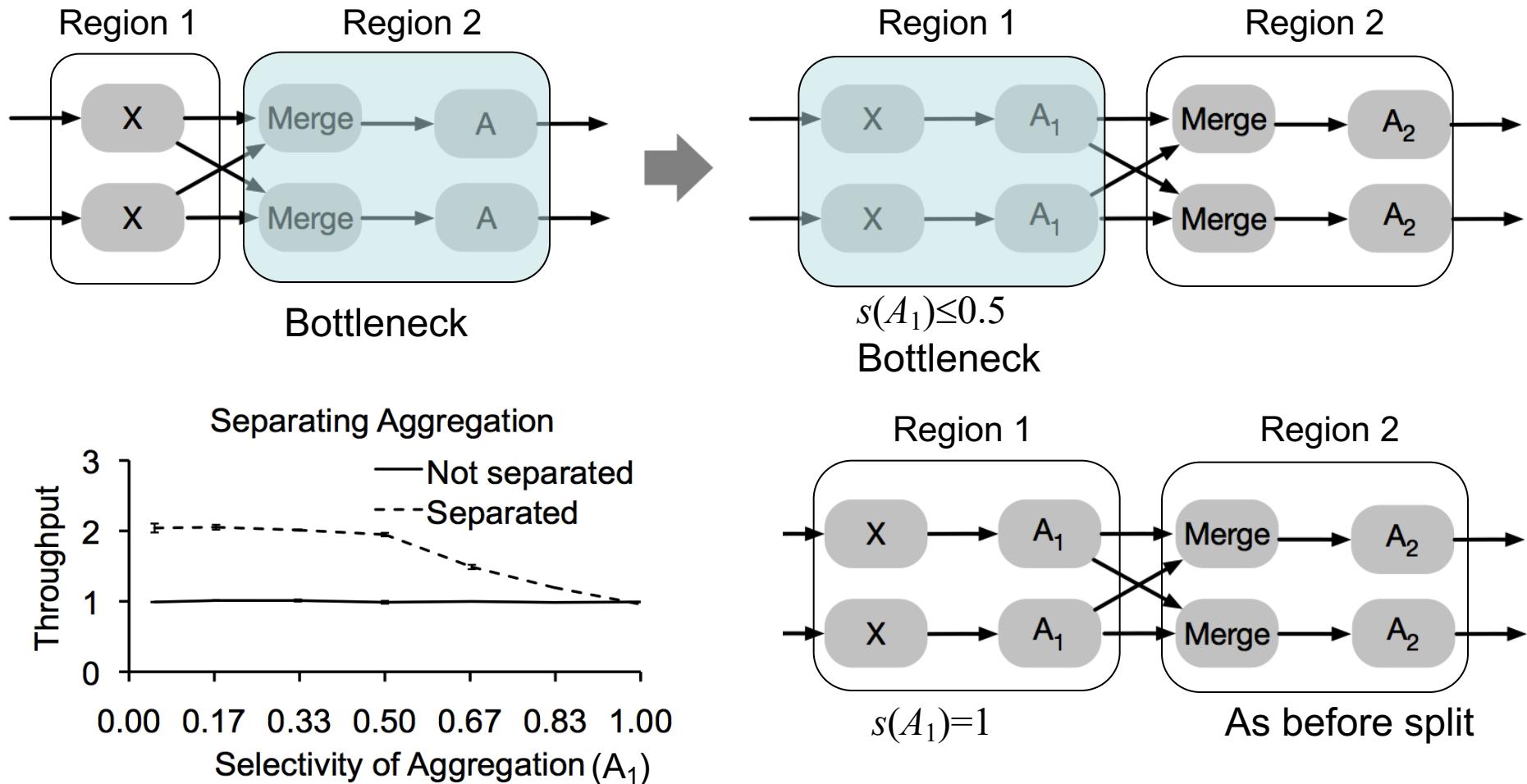
Assume A performs filtering (by sentiment and price)

Can do pre-aggregation A₁ before merge

Assume $c(A_1) = 0.5$, $c(A_2) = 0.5$



Profitability of Operator Separation



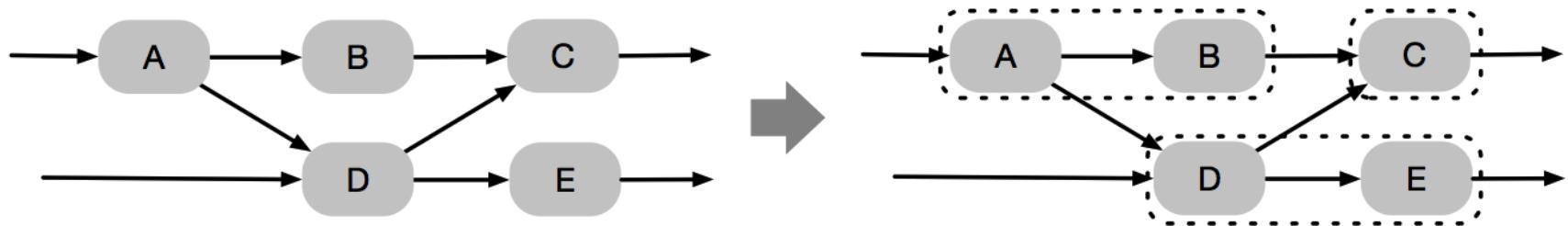
Safety of Operator Separation

- Ensure that the combination of the separated operators is equivalent to the original operator
 - $A_2(A_1(s)) = A(s)$
 - This is easier to establish in the relational domain
 - If A is a selection op and the selection predicate uses logical conjunction, A_1 and A_2 can be selections on the conjuncts
 - If A is a projection that assigns multiple attributes, then A_1 and A_2 can be projections that assign the attributes separately.
 - If A is an idempotent aggregation, then A_1 and A_2 can simply be the **same** as A itself.

Idempotence: property of certain operations that can be applied multiple times without changing the result

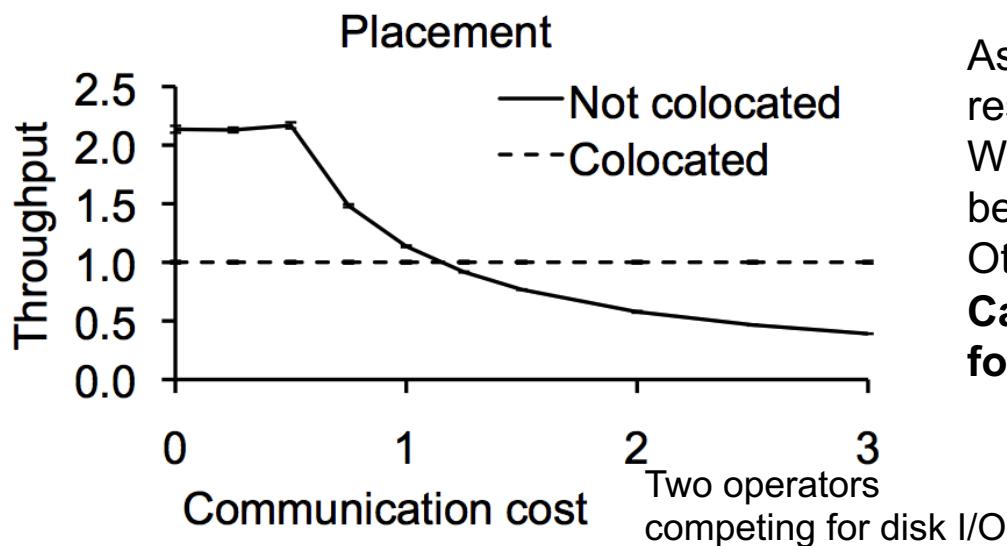
Placement

- *Assigning processing elements to hosts.*



Profitability of Placement

- Placement trades communication cost against resource utilization (disk/memory/CPU)
 - Can use more than one host, if you pay the cost of intra-host transfers



Assumption: No CPU contention, only resource contention is disk
When communication overhead low, much better to place on different machine
Otherwise degrades with communication cost
Can we generalize to cases with contention for other resources?

Placement with Scheduler

- The Streams job scheduler selects the host (node) where each PE in a submitted job is to be placed
 - Done so in accordance with any requirements specified in the application
- The host is selected from a set of candidate hosts
- Makes an attempt to balance the processing load of the hosts
 - Takes into account their current load
- What does this mean?
 - You can let the system balance the workload for you
- User can provide directives on placement to scheduler

Placement: config Clause

```
Streams<uint32 val> Fil = Filter(Beat) {  
    logic state : mutable blooean sw = false;  
    onTuple Beat : sw = !sw;  
    param filter : sw;  
    config placement: host("streams.ibm.com");  
}
```

Format	Description
host("10.4.40.24");	User picks host by IP address
host("streams.ibm.com");	User picks host by name
host(MyPool[5]);	User picks host at a fixed offset from the pool
host(MyPool);	System picks host from the pool at runtime

Placement: Host Pools

The main composite's config clause is used to create node pools

Node pools can be defined with specific associated hosts

Can use the `createPool` sub-config clause to create the pool at runtime

```
composite Main {  
    graph  
        operator1 {}  
        operator2 {}  
    config hostPool:  
        MyPool= ["ibmclass.ibm.com", "10.8.5.6"],  
        P1= createPool({size=5u, tags=["res","prod"]}, Sys.Shared),  
        P2= createPool({size=10u, tags=["my"]}, Sys.Exclusive);  
}
```

Node pools can be defined with specific associated hosts

Other Placement Directives

- hostColocation
 - Operator instances with the same hostColocation value will run on the same host

```
config placement : hostColocation("myHost");
```
- hostIsolation
 - Operator instance belongs to a partition that has a host of its own
 - Other operator instances can be in the partition
 - But no other partition can run on that host

```
config placement : hostIsolation;
```
- hostExlocation
 - Any operator instances that have the same hostExlocation value must run on different hosts

```
config placement : hostExlocation("notMyHost");
```

Safety of Placement

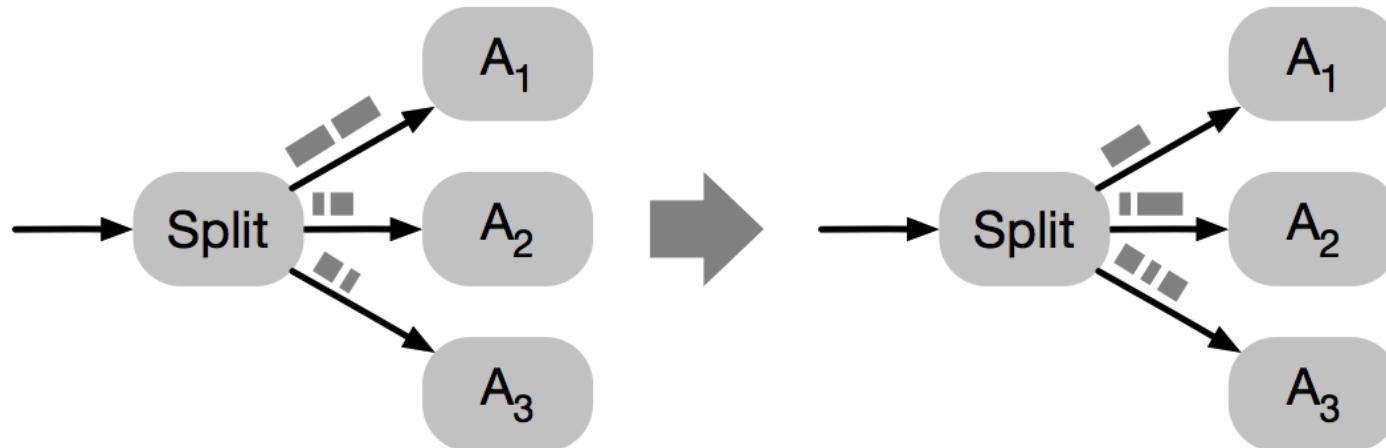
- Ensure resource kinds
 - Placement is safe if each host has the right resources for all the operators placed on it.
- Ensure resource amounts
 - Total amount of resource required by the co-placed operators should not exceed the amount of resources available on a single host
- If placement is dynamic, move only relocatable operators
 - Moving state and ensuring no tuple loss is an issue
 - Operators with OS resources such as sockets and files may be designated as non-movable.

Dynamic Placement

- Placement decisions are often made during submission time
- Some placement algorithms continue to be active after the job starts, to adapt to changes in load or resource availability
 - As discussed, this poses additional safety issues
 - State migration, tuple loss

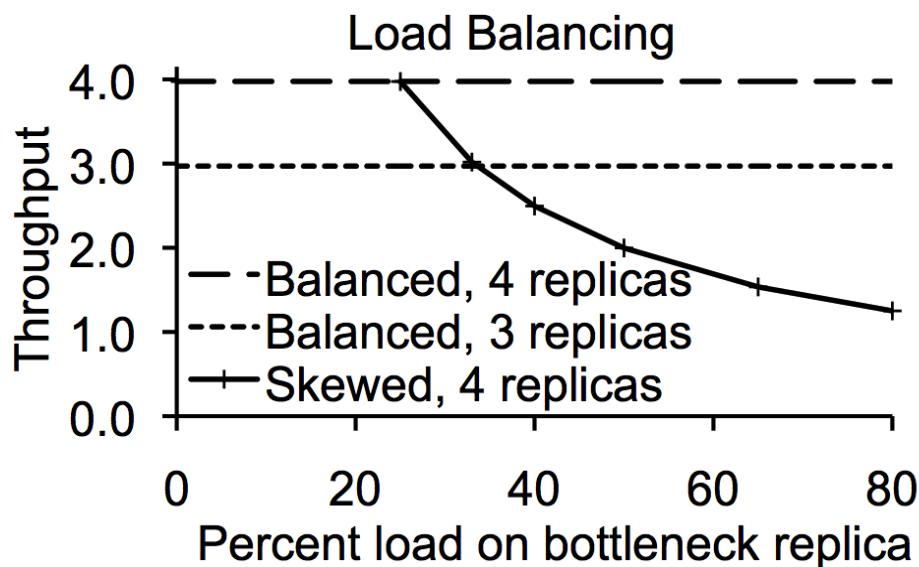
Load-balancing for Data Parallel Splits

- *Distribute workload evenly across resources*
 - *Data parallel processing*



Profitability of Load-balancing

- Load balancing is profitable if it compensates for skew
- Skew can be in the
 - data: some partitions have higher volume than others
 - resources: some channels have more capacity than others



Split operator that streams data to 3 or 4 replicated operators. With perfect load balancing, throughput is linear in replicas. Without load balancing, there is skew, and throughput is bounded by whichever replica receives the most load

Safety of Load-balancing

- Avoid starvation
 - Every data item eventually gets processed
- Each worker is qualified
 - A data item can only be routed to a worker that is capable of handling it
 - E.g. In partitioned stateful parallelism, tuples from the same partition should go to the same place

Variations in Load-balancing

- Balancing load during operator placement
 - Host resources are utilized evenly
 - During submission, prefer placement on hosts that are less utilized
- Balance load during fission
 - Apply fission and then decide how much load each parallel channel gets
 - i.e., combined fission and load balance

Dynamic Load Balancing

- We discussed two forms of load balancing
 - Routing based: Usually done dynamically
 - Requires splitter level decisions at runtime
 - Placement based: Typically static
 - Requires moving operators at runtime
 - Could take long time, may not be safe

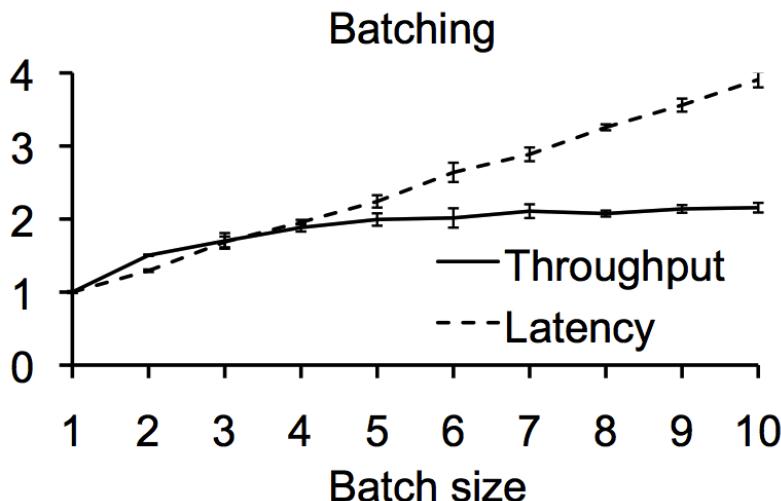
Batching

- Also known as
 - Train scheduling, execution scheduling
- *Process multiple data items in a single batch*
 - Like RDDs



Profitability of batching

- Batching trades throughput for latency
- Can reduce operator firing and communication costs
- The amortizable costs can include
 - Scheduling cost, context switch cost, synchronization cost, instruction cache costs, data cache costs



Assume $c(A) = c(\text{Tuple}) + c(\text{overhead})$
With batching n tuples
 $c(A') = n \times c(\text{Tuple}) + c(\text{overhead})$
Amortize overhead costs
Latency: delay between when tuple arrives to when it is transmitted. *Different for different tuples in the batch.* Maximum delay scales linearly with batch size.

Safety of Batching

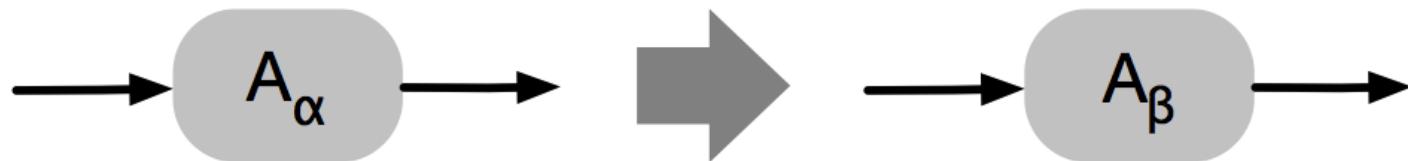
- Avoid deadlock
 - In the presence of feedback loops
 - In the presence of shared locks
- Satisfy deadlines
 - Real-time constraints on the per-tuple delay
 - QoS (quality of service) constraints, where delay impacts the QoS value
 - E.g. maintaining frame rates to avoid jitter in video processing

Dynamism

- The batch size can be set
 - Statically
 - StreamIt uses instruction cache vs data cache tradeoff to set the batch sizes
 - Dynamically
 - Aurora uses train scheduling to minimize context switching costs
 - Staged Event-Driven Architecture (SEDA) uses the tradeoff between latency and adaptivity to set the buffer sizes

Algorithm Selection

- Also known as
 - translation to physical query plan
- *Use a faster algorithm for implementing an operator.*

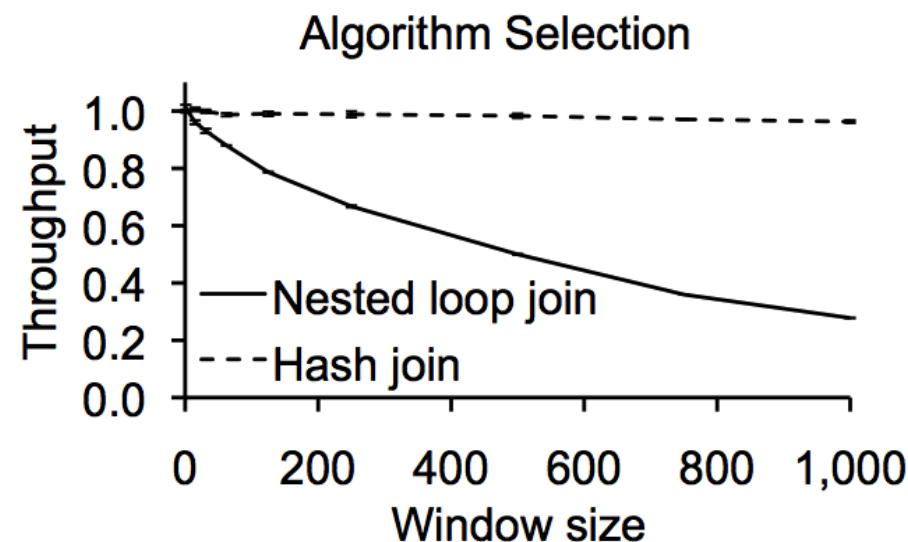
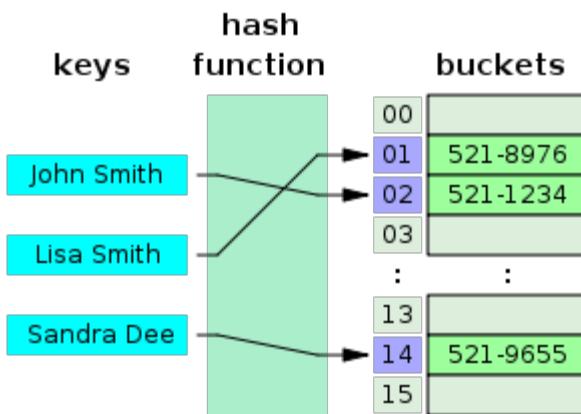


Profitability of algorithm selection

- It is profitable if a costly operator is replaced with a cheap one
- One algorithm may not be superior in all cases
 - E.g. An algorithm that works faster for small tuples vs. an algorithm that works faster for larger tuples
 - E.g. An algorithm that works faster but uses more memory.
- Example: Join
 - A hash-based implementation is almost always faster (unless the window is very small) than loop join (linear scan)
 - A hash-based implementation is suitable only for equi-joins

Profitability of algorithm selection

- Loop join: For each tuple iterate through the tuples in window of other stream to find match
- Hash join: Create a hash table for the stream window. Apply hash function to new tuple on other stream to find match



Aside on Hash Functions

```
function Hash(key)
    return key mod PrimeNumber
end

Additive Hash
ub4 additive(char *key, ub4 len, ub4 prime)
{
    ub4 hash, i;
    for (hash=len, i=0; i<len; ++i)
        hash += key[i];
    return (hash % prime);
}

Rotating Hash
ub4 rotating(char *key, ub4 len, ub4 prime)
{
    ub4 hash, i;
    for (hash=len, i=0; i<len; ++i)
        hash = (hash<<4)^(hash>>28)^key[i];
    return (hash % prime);
}
```

```
Bernstein's hash
ub4 bernstein(ub1 *key, ub4 len, ub4 level)
{
    ub4 hash = level;
    ub4 i;
    for (i=0; i<len; ++i) hash = 33*hash + key[i];
    return hash;
}
```

If your keys are lowercase English words, this will fit 6 characters into a 32-bit hash with no collisions

Several cryptographic hash functions, e.g. SHA2

Safety of algorithm selection

- Safety
 - Ensure same behavior
 - Be aware of variations
- Variations
 - Physical query plans
 - Auto-tuners
 - Empirical optimization
 - Different semantics
 - Load shedding via cheaper but less accurate algorithms

Dynamics of algorithm selection

- Dynamism
 - Algorithm selection can be dynamic
- Both algorithms are provisioned and the right one is selected at runtime
 - Performed via dynamic routing
- Change algorithm operation via parameter choice
 - E.g. size of ensemble (IMARS example)

Load-shedding

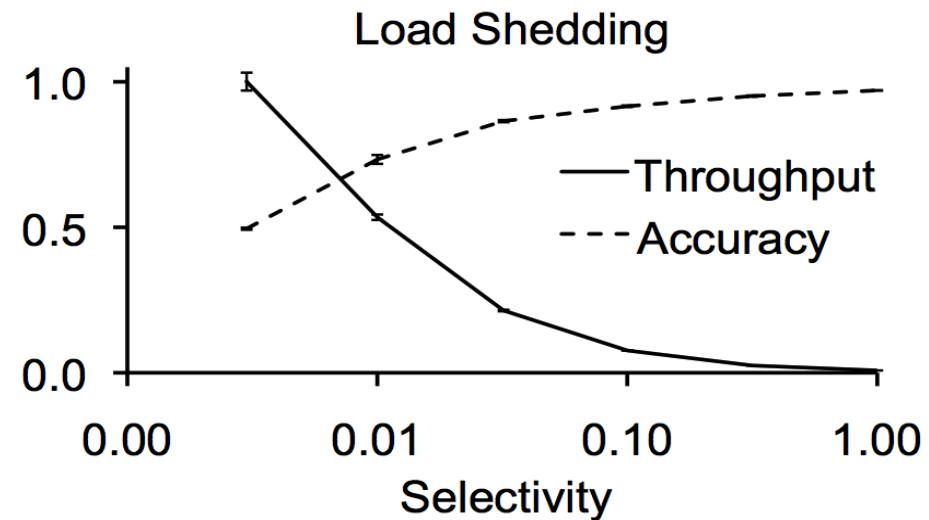
- Also known as
 - Admission control, graceful degradation
- *Degrade gracefully when overloaded.*



Profitability of load-shedding

- Load shedding improves throughput at the cost of accuracy
- Consider an aggregator-like algorithm that constructs a histogram
 - Sampling can be used effectively
 - Reducing the rate to one tenth may have a negligible impact on the histogram accuracy

Euclidean distance between histogram at full accuracy, and histogram at reduced accuracy

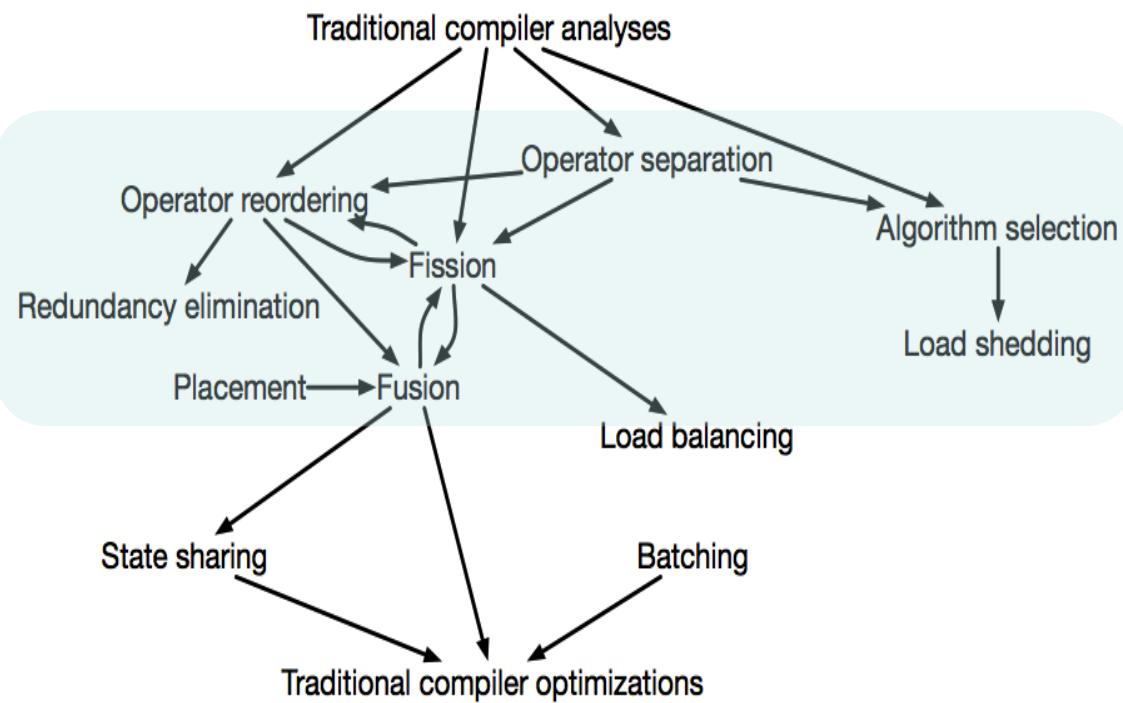


Safety and Dynamism of load-shedding

- Load shedding, by definition, is not safe
 - Ideally, the reduction in the quality should be acceptable
 - Major area of research
- Load shedding is always dynamic
 - Change the selectivity based on current load

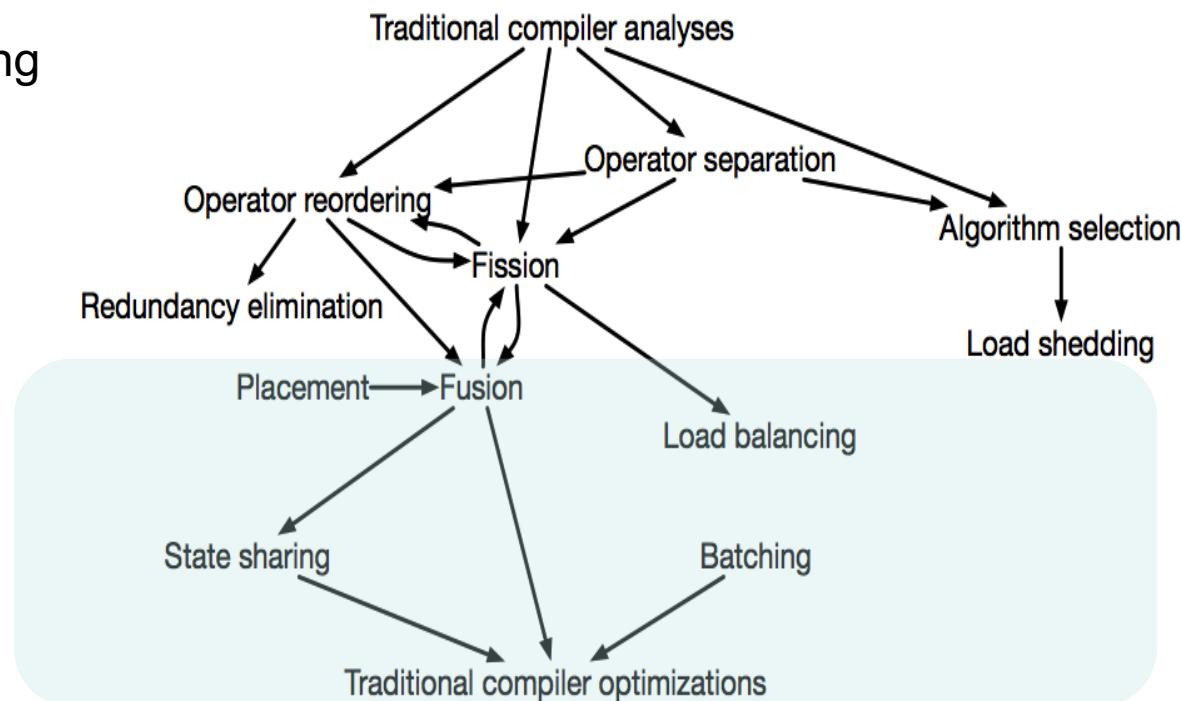
Interactions between Optimizations

1. Primary enablers are operator separation and operator reordering
2. Circular enablement between operator reordering and fission
3. Circular enablement between fission and fusion
4. Fission makes it easier to balance load



Interactions between Optimizations

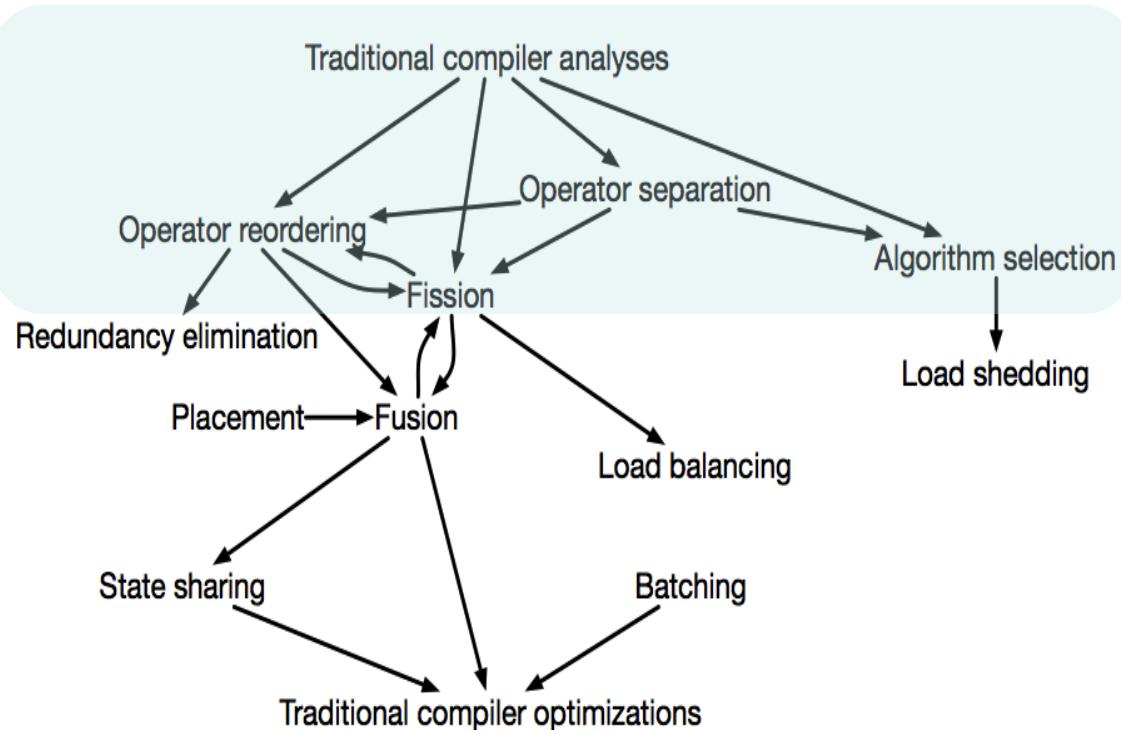
1. Fusion enables function inlining
2. State sharing enables scalar replacement (replace arrays/structs with scalars), if the compiler can statically determine the size of a communication queue based on fixed data rates, and can determine the index of data access for each iteration
3. Batching enables loop unrolling, software pipelining, and loop optimizations



<http://www.cs.uiuc.edu/class/fa06/cs498dp/notes/optimizations.pdf>

Interactions between Optimizations

1. Operator reordering can be enabled by commutativity analysis
2. Operator separation can be supported by compiler analysis
3. Fission can also be supported by compiler analysis
4. Algorithm selection can be supported by worst-case execution time analysis



Discussion on Optimization

- Metrics for evaluation
 - many ways to measure whether a streaming optimization was profitable
 - throughput, latency, quality of service (QoS), accuracy, power, and system utilization
- Need standard benchmarks for streaming workloads
 - Stanford stream query repository including Linear Road [Arasu et al. 2006],
 - BiCEP benchmarks [Mendes et al. 2009]
 - StreamIt benchmarks [Thies and Amarasinghe 2010]
 - *Project idea?* ☺

Arasu, A., Babu, S., and Widom, J. 2006. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal* 15, 2 (June), 121–142.

Mendes, M. R. N., Bizarro, P., and Marques, P. 2009. A performance study of event processing systems. In *TPC Technology Conference on Performance Evaluation & Benchmarking (TPC TC)*. 221–236.

Thies, W. and Amarasinghe, S. 2010. An empirical characterization of stream programs and its implications for language and compiler design. In *Parallel Architectures and Compilation Techniques (PACT)*. 365–376.

Wrap-up on Optimization

- Non-trivial to optimize performance
 - Compiler and programming language support important
 - Often requires manual tuning and experimentation
 - Lots of dynamics (compute resources, operator performance, data characteristics and workload)
 - Several different competing objectives
- Guiding principle
 - Keep operators small and lightweight and explore many knobs
- Impact on fault tolerance
 - Many optimizations are orthogonal to whether or not the system is fault tolerant.
- Centralized versus distributed optimization
 - Assumptions on shared memory, or other resources may not be valid

Introduction to Streaming Analytics

Objectives

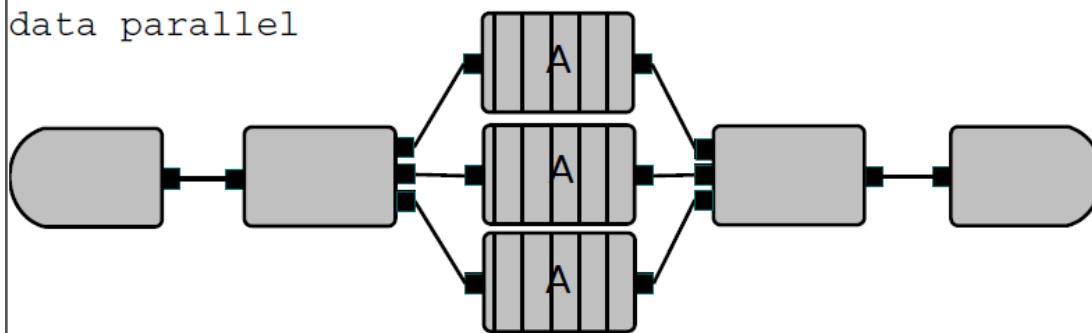
- Streaming Optimizations
 - Recap from last lecture
- Logistics
 - Homework 2
 - Seminar
- Analytics
 - Data Preprocessing I

Recap: Types of Parallelism

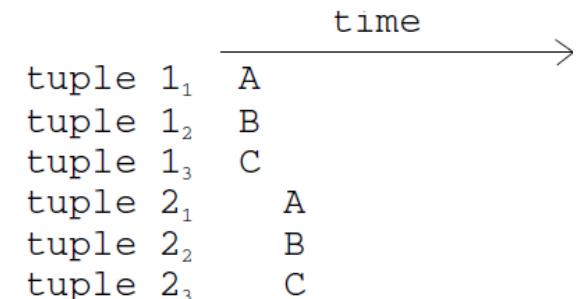
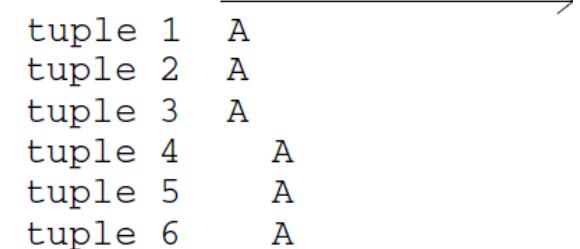
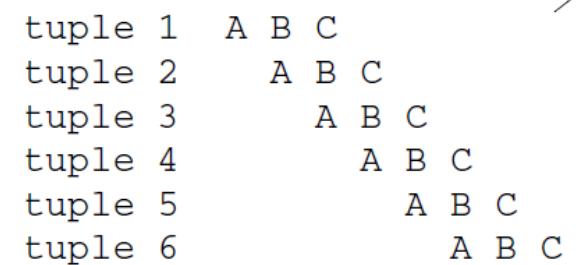
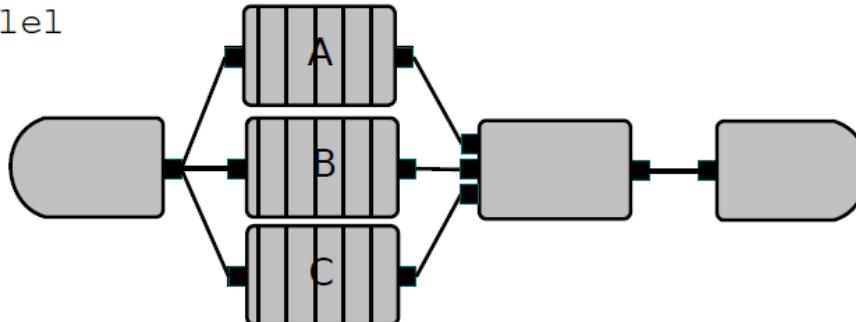
pipelined parallel



data parallel



task parallel

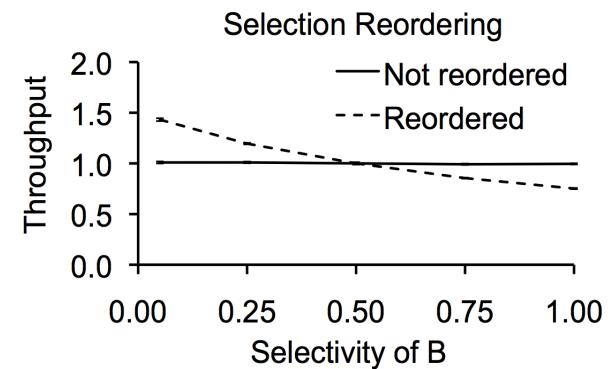
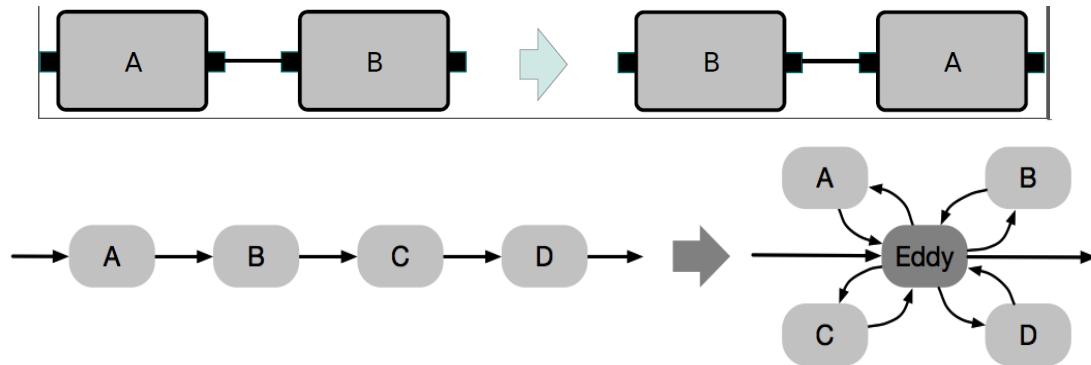


Recap: Kinds of Optimization

- Three dimensions
 - Does it change the graph?
 - Optimizations that involve graph transformations change the graph.
E.g.: Fission
 - Some optimizations do not require transformation. E.g. Load balancing.
 - Does it impact the application semantics?
 - Ideally, an optimization should not impact the application semantics.
 - There are a few exceptions: load shedding, algorithm selection
 - Is it dynamic?
 - If the optimization can be applied at runtime, potentially based on current resource and workload availability, then it is dynamic
 - Static optimizations are often done at compile-time

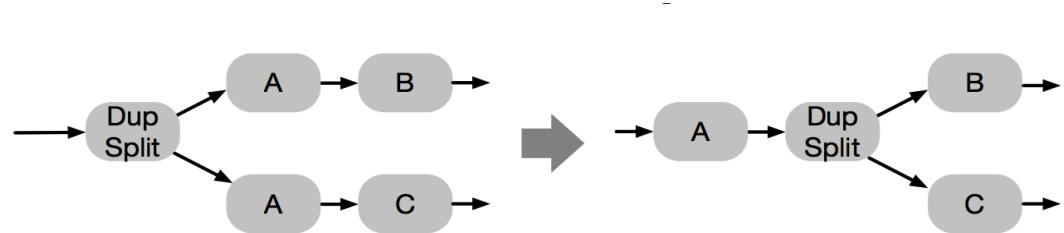
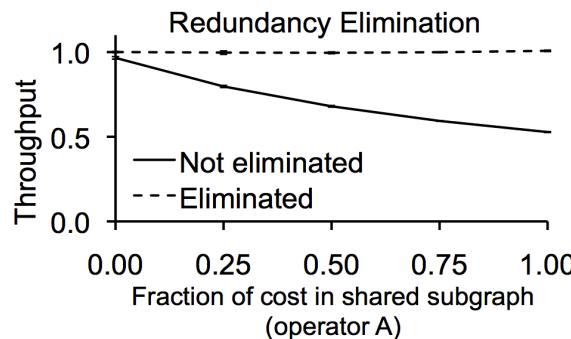
Operator Reordering

- Profitable
 - When selectivity value of second operator smaller than first
- Safety
 - Ensure commutativity



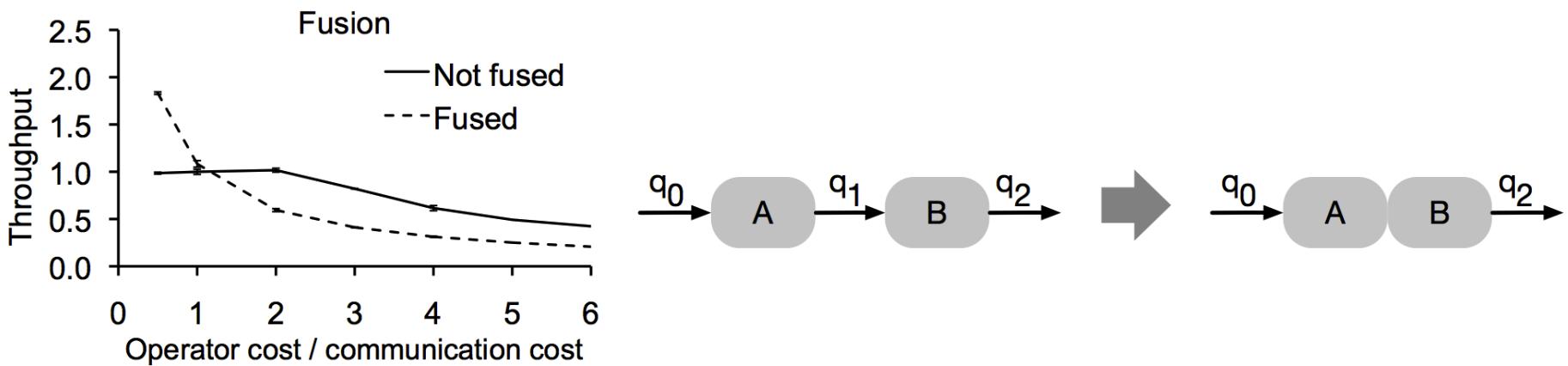
Redundancy Elimination

- Profitable
 - When cost of replicated operator is major fraction of total cost
- Safety
 - Make sure state is handled correctly, and that the operators are actually the same



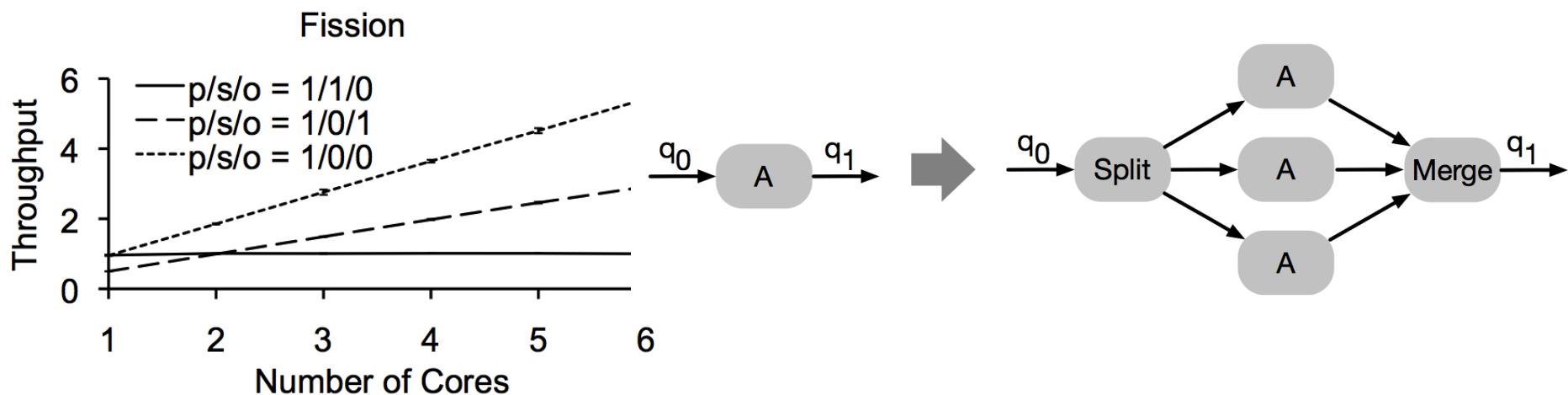
Fusion

- Profitable
 - When communication overhead is larger than processing
- Safety
 - Ensure resources are available, and have capacity



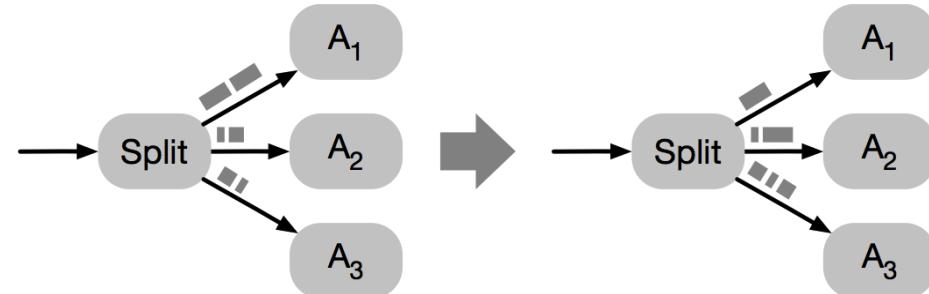
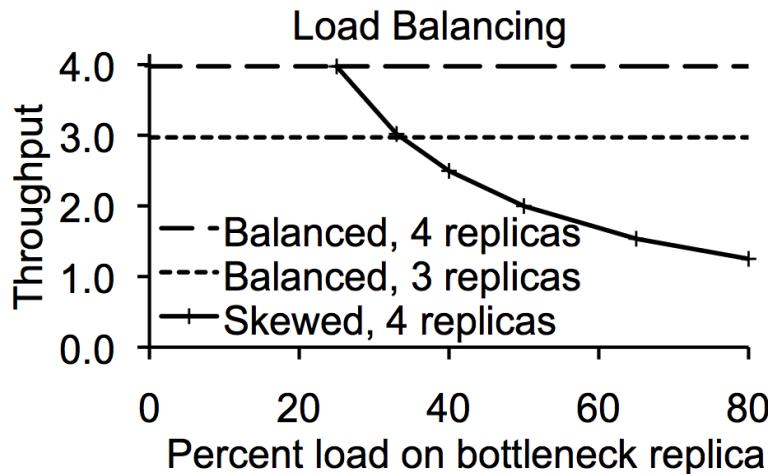
Fission: Data Parallelism

- Profitable
 - Trades split/merge overheads against resource utilization
- Safety
 - Ensure necessary resources available, stateful partitioning



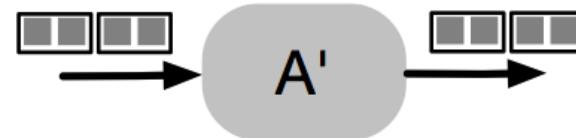
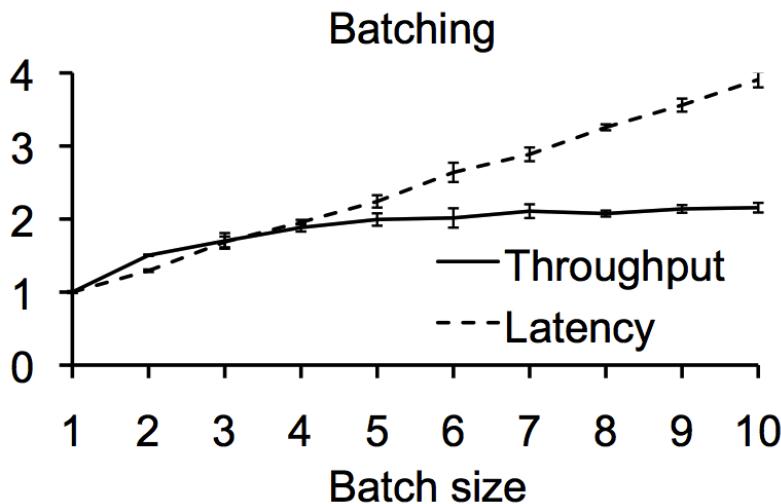
Load-balancing for Data Parallel Splits

- Profitable
 - Need to compensate for workload skew
- Safety
 - Make sure to avoid starvation. Manage state appropriately



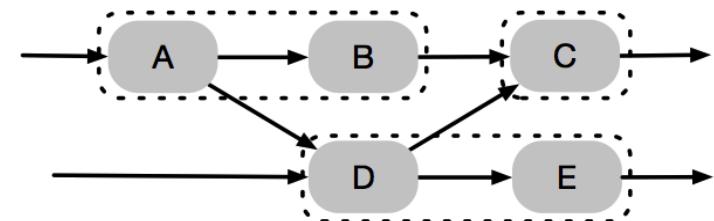
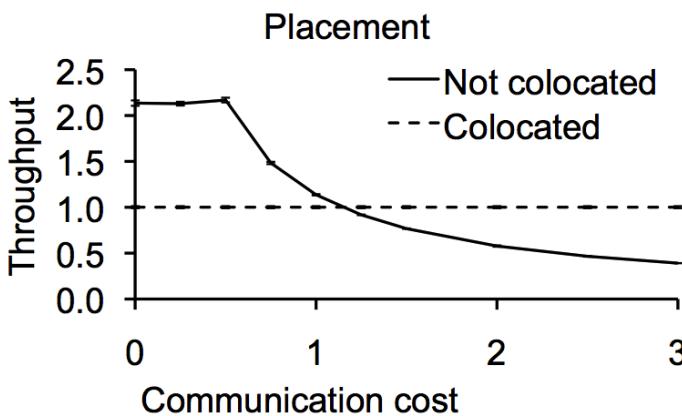
Batching

- Profitable
 - trades throughput for latency, amortizes overheads
- Safety
 - Avoid deadlocks, and satisfy real-time constraints



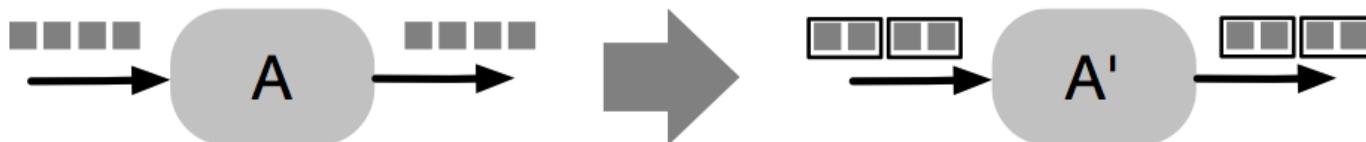
Placement

- Profitable
 - Trades communication cost against resource utilization
- Safety
 - Ensure resource kinds and availability



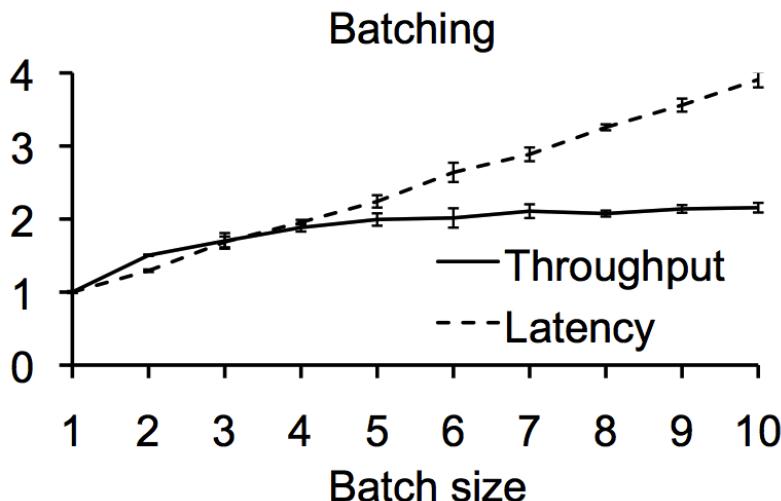
Batching

- Also known as
 - Train scheduling, execution scheduling
- *Process multiple data items in a single batch*
 - Like RDDs



Profitability of batching

- Batching trades throughput for latency
- Can reduce operator firing and communication costs
- The amortizable costs can include
 - Scheduling cost, context switch cost, synchronization cost, instruction cache costs, data cache costs



Assume $c(A) = c(\text{Tuple}) + c(\text{overhead})$
With batching n tuples
 $c(A') = n \times c(\text{Tuple}) + c(\text{overhead})$
Amortize overhead costs
Latency: delay between when tuple arrives to when it is transmitted. *Different for different tuples in the batch.* Maximum delay scales linearly with batch size.

Safety of Batching

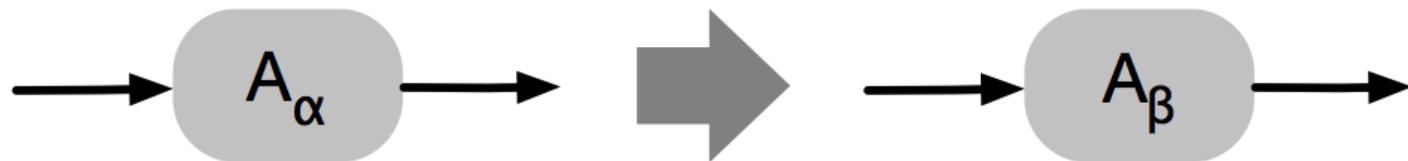
- Avoid deadlock
 - In the presence of feedback loops
 - In the presence of shared locks
- Satisfy deadlines
 - Real-time constraints on the per-tuple delay
 - QoS (quality of service) constraints, where delay impacts the QoS value
 - E.g. maintaining frame rates to avoid jitter in video processing

Dynamism

- The batch size can be set
 - Statically
 - StreamIt uses instruction cache vs data cache tradeoff to set the batch sizes
 - Dynamically
 - Aurora uses train scheduling to minimize context switching costs
 - Staged Event-Driven Architecture (SEDA) uses the tradeoff between latency and adaptivity to set the buffer sizes

Algorithm Selection

- Also known as
 - translation to physical query plan
- *Use a faster algorithm for implementing an operator.*

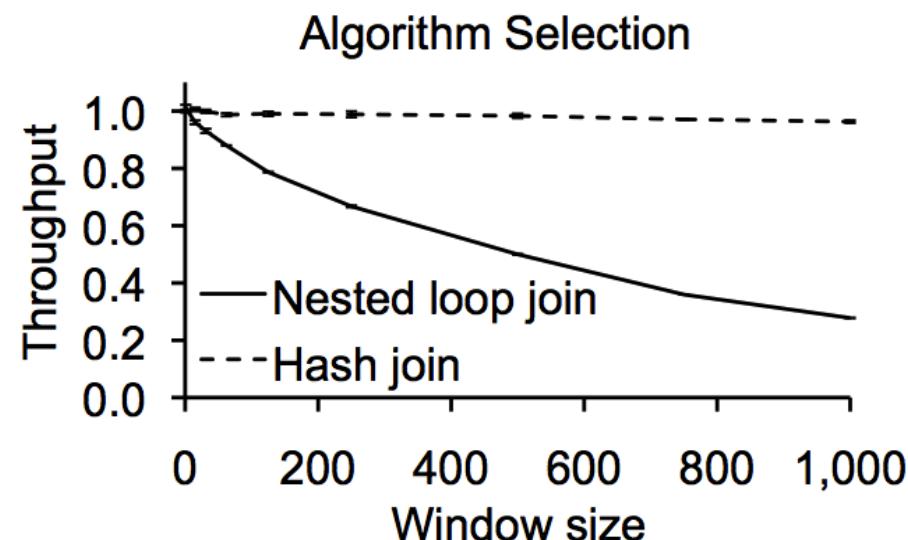
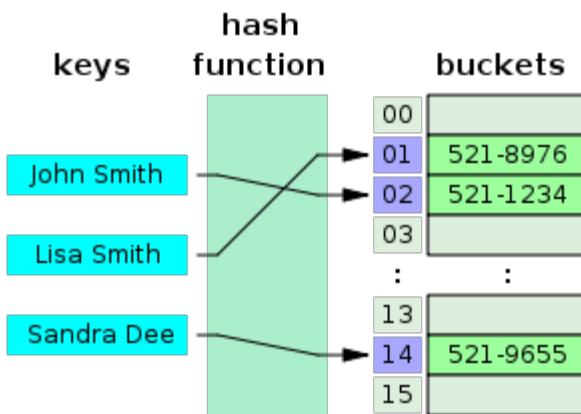


Profitability of algorithm selection

- It is profitable if a costly operator is replaced with a cheap one
- One algorithm may not be superior in all cases
 - E.g. An algorithm that works faster for small tuples vs. an algorithm that works faster for larger tuples
 - E.g. An algorithm that works faster but uses more memory.
- Example: Join
 - A hash-based implementation is almost always faster (unless the window is very small) than loop join (linear scan)
 - A hash-based implementation is suitable only for equi-joins

Profitability of algorithm selection

- Loop join: For each tuple iterate through the tuples in window of other stream to find match
- Hash join: Create a hash table for the stream window. Apply hash function to new tuple on other stream to find match



Aside on Hash Functions

```
function Hash(key)
    return key mod PrimeNumber
end

Additive Hash
ub4 additive(char *key, ub4 len, ub4 prime)
{
    ub4 hash, i;
    for (hash=len, i=0; i<len; ++i)
        hash += key[i];
    return (hash % prime);
}

Rotating Hash
ub4 rotating(char *key, ub4 len, ub4 prime)
{
    ub4 hash, i;
    for (hash=len, i=0; i<len; ++i)
        hash = (hash<<4)^(hash>>28)^key[i];
    return (hash % prime);
}
```

```
Bernstein's hash
ub4 bernstein(ub1 *key, ub4 len, ub4 level)
{
    ub4 hash = level;
    ub4 i;
    for (i=0; i<len; ++i) hash = 33*hash + key[i];
    return hash;
}
```

If your keys are lowercase English words, this will fit 6 characters into a 32-bit hash with no collisions

Several cryptographic hash functions, e.g. SHA2

Safety of algorithm selection

- Safety
 - Ensure same behavior
 - Be aware of variations
- Variations
 - Physical query plans
 - Auto-tuners
 - Empirical optimization
 - Different semantics
 - Load shedding via cheaper but less accurate algorithms

Dynamics of algorithm selection

- Dynamism
 - Algorithm selection can be dynamic
- Both algorithms are provisioned and the right one is selected at runtime
 - Performed via dynamic routing
- Change algorithm operation via parameter choice
 - E.g. size of ensemble (IMARS example)

Load-shedding

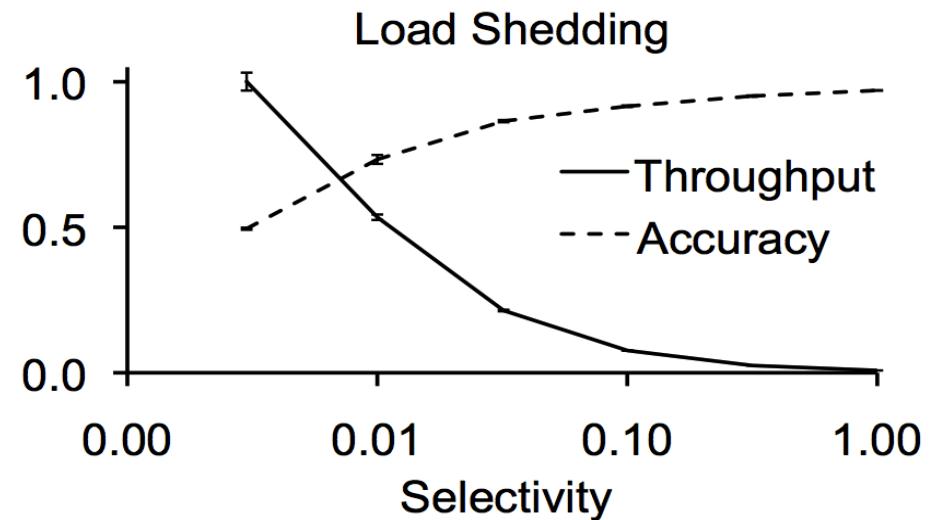
- Also known as
 - Admission control, graceful degradation
- *Degrade gracefully when overloaded.*



Profitability of load-shedding

- Load shedding improves throughput at the cost of accuracy
- Consider an aggregator-like algorithm that constructs a histogram
 - Sampling can be used effectively
 - Reducing the rate to one tenth may have a negligible impact on the histogram accuracy

Euclidean distance between histogram at full accuracy, and histogram at reduced accuracy



Safety and Dynamism of load-shedding

- Load shedding, by definition, is not safe
 - Ideally, the reduction in the quality should be acceptable
 - Major area of research
- Load shedding is always dynamic
 - Change the selectivity based on current load

Recap Optimization Catalog

Optimization	Graph	Semantics	Dynamics
Operator re-ordering	changed	unchanged	(depends)
Redundancy elimination	changed	unchanged	(depends)
Operator separation	changed	unchanged	static
Fusion	changed	unchanged	(depends)
Fission	changed	(depends)	(depends)
Placement	unchanged	unchanged	(depends)
Load balancing	unchanged	unchaged	(depends)
State sharing	unchanged	unchanged	static
Batching	unchanged	unchanged	(depends)
Algorithm selection	unchanged	(depends)	(depends)
Load shedding	unchanged	changed	dynamic

Discussion on Optimization

- Metrics for evaluation
 - many ways to measure whether a streaming optimization was profitable
 - throughput, latency, quality of service (QoS), accuracy, power, and system utilization
- Need standard benchmarks for streaming workloads
 - Stanford stream query repository including Linear Road [Arasu et al. 2006],
 - BiCEP benchmarks [Mendes et al. 2009]
 - StreamIt benchmarks [Thies and Amarasinghe 2010]
 - *Project idea?* ☺

Arasu, A., Babu, S., and Widom, J. 2006. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal* 15, 2 (June), 121–142.

Mendes, M. R. N., Bizarro, P., and Marques, P. 2009. A performance study of event processing systems. In *TPC Technology Conference on Performance Evaluation & Benchmarking (TPC TC)*. 221–236.

Thies, W. and Amarasinghe, S. 2010. An empirical characterization of stream programs and its implications for language and compiler design. In *Parallel Architectures and Compilation Techniques (PACT)*. 365–376.

Wrap-up on Optimization

- Non-trivial to optimize performance
 - Compiler and programming language support important
 - Often requires manual tuning and experimentation
 - Lots of dynamics (compute resources, operator performance, data characteristics and workload)
 - Several different competing objectives
- Guiding principle
 - Keep operators small and lightweight and explore many knobs
- Impact on fault tolerance
 - Many optimizations are orthogonal to whether or not the system is fault tolerant.
- Centralized versus distributed optimization
 - Assumptions on shared memory, or other resources may not be valid

Stream Mining and Analytics

- Streaming analytics
 - Techniques and algorithms from fields such as data mining, machine learning, statistics, signal processing, and artificial intelligence that are adapted to work in a streaming setting
- Important features of the streaming setting
 - Ideally Single scan: Data is seen only once, in order of arrival
 - Unlike stored data, that can be scanned multiple times, potentially indexed and accessed in other ways
 - Limited memory: The amount of memory that can be used is small compared to the size of the data stream
 - Linear space algorithms may be too expensive
 - Logarithmic or constant is good
 - Distributed approaches for mining and analysis can be exploited

Outline of Data Mining Process

- Data acquisition
 - Collect data from external sources
- Data pre-processing
 - Prepare data for further analysis: data cleaning, data interpolation (missing data), data normalization (heterogeneous sources), temporal alignment and data formatting, data reduction
- Data transformation
 - Select an appropriate representation for the data
 - Select or compute the relevant features (*feature extraction/selection*)
- Modeling (data mining)
 - Identify interesting patterns, similarity and groupings, partition into classes, fit functions, find dependencies and correlations, identifying abnormal data
- Evaluation
 - Use of the mining model and evaluation of the results

Data Preprocessing and Transformation

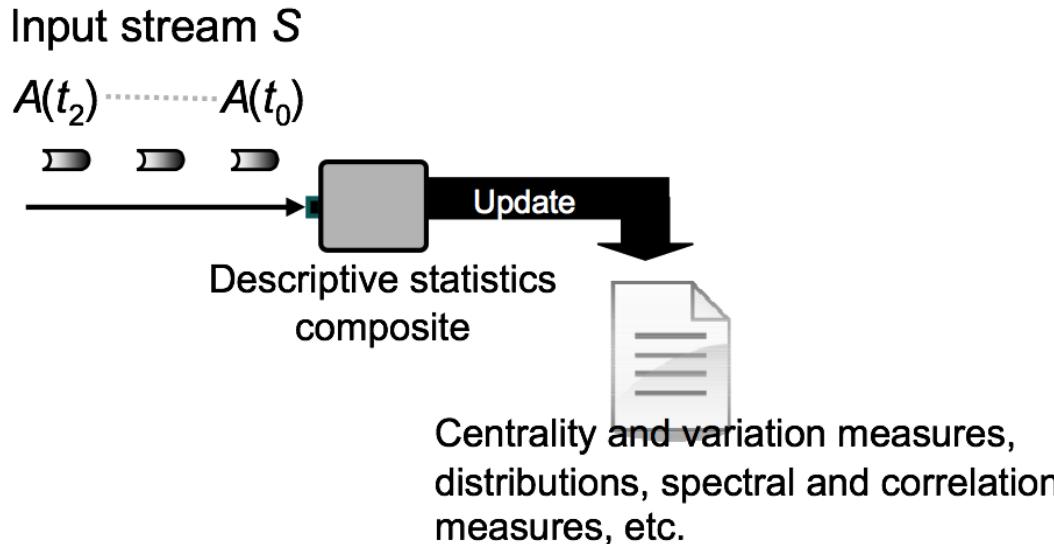
- Descriptive statistics
 - Extracting simple quantitative statistics of the distribution
- Sampling
 - Reducing the volume of the input data by retaining only an appropriate subset for analysis
- Sketches
 - Compact data structures that contain synopses of the streaming data, for approximate query answering
- Quantization
 - Reduce the fidelity of individual data samples to lower compute and memory costs
- Dimensionality reduction
 - Reduce the number of attributes within each tuple to decrease data volume and improve accuracy of the models
- Transforms
 - Convert data items or tuples and their attributes from one domain to another, such that data is better suited for further analysis

Quick Aside: Notation

- Stream: Sequence of tuples
 - $S = A(t_0), A(t_1), A(t_2), \dots$
- Tuple: Set of N attributes
 - $A(t) = \{a_0(t), a_1(t), \dots, a_{N-1}(t)\}$
- Special class of tuples: Numeric tuples
 - Common in many applications
 - Methods to convert categorical values to numeric values (e.g. bag of words)
 - Treated as vectors

$$A(t) = \mathbf{x}(t) = \begin{bmatrix} x_0(t) \\ \vdots \\ x_{N-1}(t) \end{bmatrix}$$

Descriptive Statistics



- Extract information from the tuples in a stream such that
 - statistical properties of the stream can be characterized or summarized
 - the quality of the data it transports assessed

Descriptive Statistics: Measures of Centrality

- Sample moments

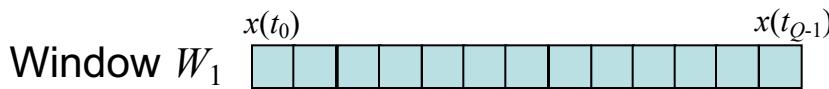
- i -th moment m_i

$$m_i = \frac{1}{Q} \sum_{j=0}^{Q-1} (x(t_j))^i$$

- First moment: Sample mean
- Second moment sample variance $\sigma^2 = m_2 - m_1^2$
- Third moment: sample skew
- Streaming friendly
 - All these can be computed using a single scan
 - They can also be computed over a sliding window

Maintaining Moments across Sliding Windows

- Moments can be computed incrementally without storing original value
 - Maintain sum (raised to the appropriate power) and number of values
- What happens with tumbling windows?
 - Reset sum and counter to 0 every tumbling window boundary
- What happens with sliding windows?
 - Consider the example of the sample mean, and a slide of 1



Sample mean: $m_1(W_1) = \frac{1}{Q} \sum_{j=0}^{Q-1} x(t_j)$



$$m_1(W_2) = \frac{1}{Q} \sum_{j=1}^Q x(t_j)$$

Naïve

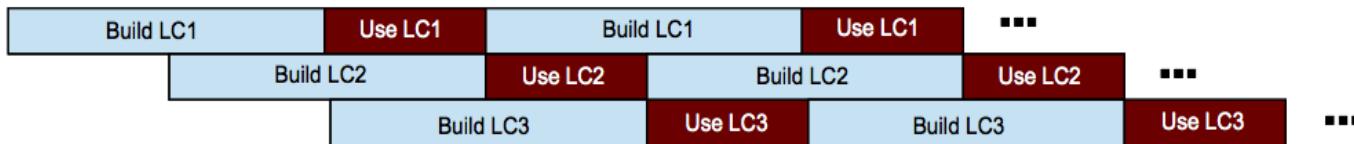
$$m_1(W_2) = m_1(W_1) - \frac{x(t_0)}{Q} + \frac{x(t_Q)}{Q}$$

Incremental

Incremental requires $O(2)$ compute, as opposed to $O(Q)$ compute, and both approaches require storing $O(Q)$ values. Note that if slide = k , we need to typically store $O(Q/k)$ values.

Maintaining Statistics Across Sliding Windows

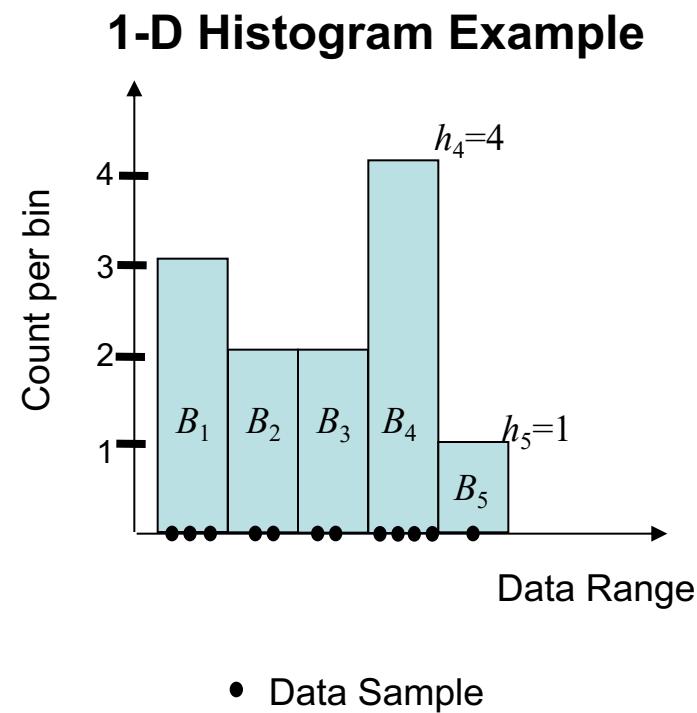
- Say you have some statistic for which you have an efficient algorithm (logarithmic space, one-scan, etc.) to compute it
- In this course, we will see several such algorithms
 - Often these algorithms assume a single scan, where data items are being added
 - However, you may need to compute sliding window versions of these to capture changing statistics
 - You will find algorithms in the literature that are specialized for this, but often the sliding window versions are complex
- A quick and dirty alternative is to use overlapping tumbling windows



- If we use n tumbling windows each of size Q
 - At any time we are using a summary computed from
 - last Q to $Q + Q/(n-1)$ items

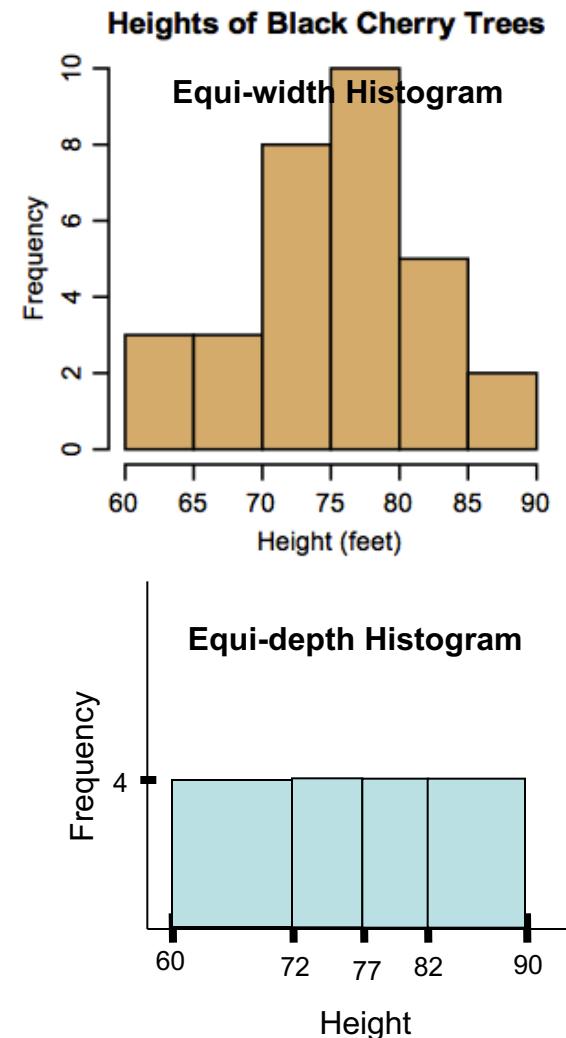
Counts and Histogram

- Given N data items
 - data items $\mathbf{x}(t_0) \dots \mathbf{x}(t_{N-1})$
- The domain of a variable is divided into q non-overlapping bins
 - B_1 through B_q
- The number of items in each bin is counted
 - h_1, \dots, h_q
 - $h_i = |\{\mathbf{x}(t_j) \in B_i ; 0 \leq j < N\}|$
- Histogram H is a collection of these counts
 - May be viewed as a vector



Counts and Histograms: 1-D

- An *equi-width histogram* has equally sized bins
 - Easy to compute using a single scan
 - Compute bucket and increment
- An *equi-depth histogram* has \sim equal count bins
 - Variable-width bins to get similar counts
 - Can be used to compute *quantiles*
 - k th q -quantile is the largest value v such that the probability of the variable being less than v is at most k/q .
 - Not easy to compute in a streaming fashion
 - Approximate techniques exist in the literature



Streaming Descriptive Statistics Algorithms

- Non sliding window methods
 - Auto Regressive Integrated Moving Average (ARIMA)
 - Kalman Filters
 - Holts-Winters Models (additive and multiplicative)
 - Kernel smoothing methods
- Sliding window methods
 - Variance, median and quantile estimation
 - Histogram Estimation

BasicCounting Algorithm

- Question: Assume you have a stream that contains tuples that are either 0s or 1s, how can you maintain the number of 1s in the last W tuples
 - Sliding window of size W , slide 1
- Assume W is large
 - So $O(W)$ is too large. We want to store logarithmic space
- The *BasicCounting* algorithm
 - Uses space of the order $O\left(\frac{1}{\varepsilon} \log^2 W\right)$
 - With error tolerance ε , i.e. the counts are within $1 \pm \varepsilon$ of its actual value. ε is specified by application objective
- Study: Data structure, update, query

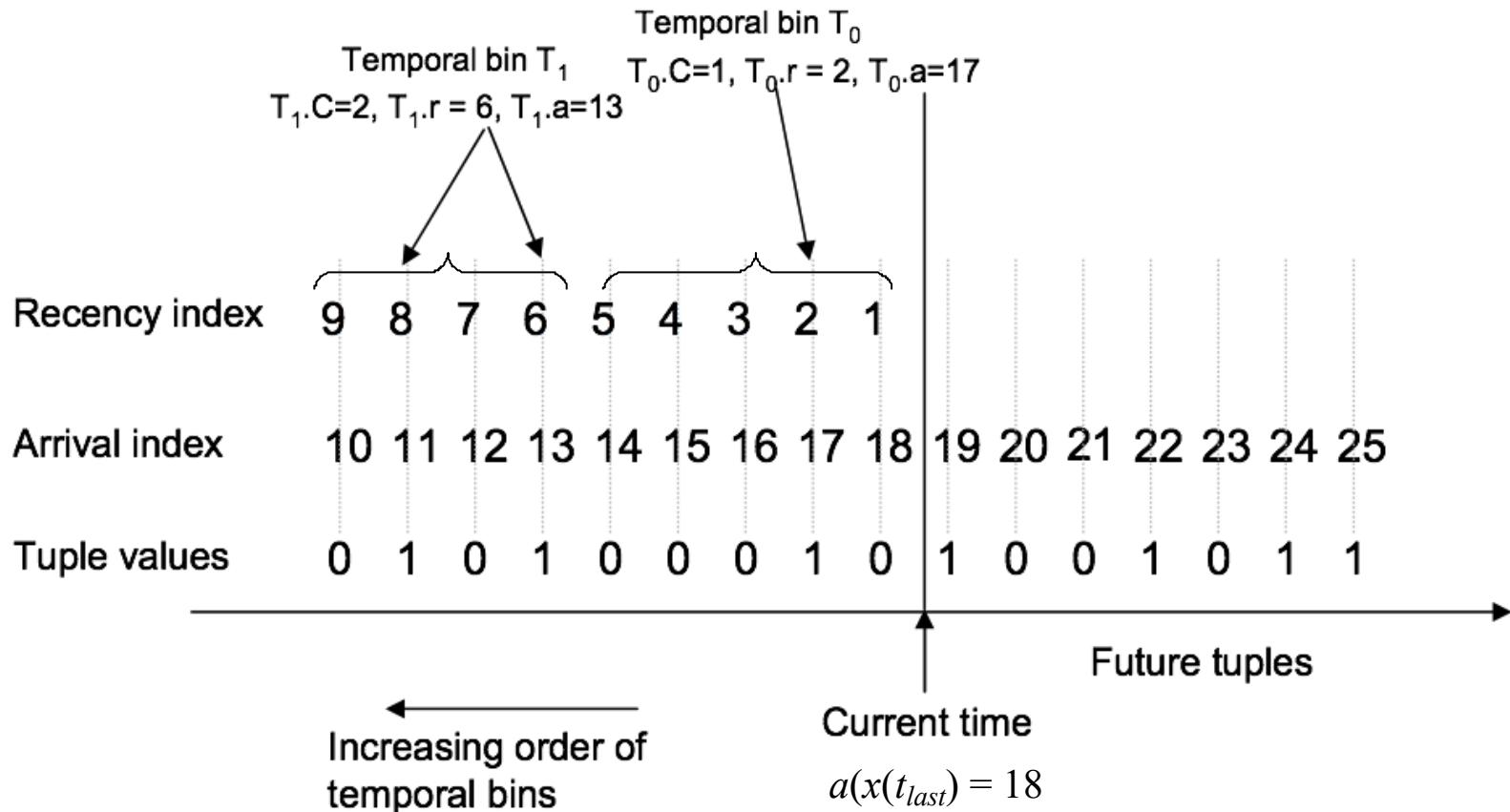
Data Structure for BasicCounting

- Assign each scalar tuple $x(t)$ an arrival index a
 - Sequence number incremented by one per tuple
- For each tuple $x(t)$ define recency index r
 - $r(x(t)) = a(x(t_{last})) - a(x(t)) + 1$
 - Distance of arrival index of tuple from last received tuple, i.e. how old or recent the tuple is
 - Changes every time a new tuple arrives!
- Maintain m temporal bins T_0, \dots, T_{m-1}
 - Each bin is a stream sub-sequence summary

Data Structure for BasicCounting

- For each bin T_i store
 - $T_i.r$: Smallest recency index of tuple that had value 1
 - $T_i.a$: Arrival index of the most recent tuple that had value 1
 - $T_i.C$: The number of tuples with value 1 in the bin
- Keep arrival index of the last received tuple as well
- Note: We do not need to explicitly store recency index for the bin
 - Can be computed from bin arrival index and arrival index of last tuple

Data Structure for BasicCounting



Structure Maintenance Algorithm

- Set $k = \left\lceil \frac{1}{\varepsilon} \right\rceil$
- When a tuple arrives, update arrival index of last tuple
 - Assume most recent bin is T_0 and oldest bin is T_{m-1}
 - If $T_{m-1}.r > W$, discard this bin
 - Does not contribute to the total count of 1s in the window
- If this tuple $x(t_{last})=1$
 - Create a new bin
 - If there are more than $k+1$ bins with count (C) 1, merge the oldest 2 bins with $C=1$ into a new bin with $C=2$
 - If there are more than $k/2+1$ bins with $C=2$, merge the oldest 2 bins with $C=2$ into a new bin with $C=4$
 - If there are more than $k/2+1$ bins with $C=y$, merge the oldest 2 bins of with $C=y$ into a new bin with $C=2*y$
- If tuple $x(t_{last})=0$, do nothing
- Creates bins with counts powers of 2 (what happens to the size?). Also ensures $T_j.C \geq T_{j-1}.C$ for all j

Structure Maintenance Example

i	$x(t_i)$	m	$T_0.C$	$T_1.C$	$T_2.C$	$T_3.C$	$T_4.C$	$T_5.C$	Operation
1	1	1	1	-	-	-	-	-	Insert
2	1	2	1	1	-	-	-	-	Insert
3	1	3	1	1	1	-	-	-	Insert
4	1	4	1	1	1	1	-	-	Insert
4	1	3	1	1	2	-	-	-	Merge
5	1	4	1	1	1	2	-	-	Insert
6	1	5	1	1	1	1	2	-	Insert
6	1	4	1	1	2	2	-	-	Merge
7	1	5	1	1	1	2	2	-	Insert
8	1	6	1	1	1	1	2	2	Insert
8	1	5	1	1	2	2	2	-	Merge
8	1	4	1	1	2	4	-	-	Merge

$$k = 2 \ (\epsilon = 0.5) \quad \text{Each bin has at least one 1}$$

Query Result and Error Analysis

- When asked for number of 1s in last W tuples, return
$$\sum_{j=0}^{m-2} T_j.C + \frac{T_{m-1}.C}{2}$$
- Error Analysis
 - No bin summarizes only tuples older than W
 - We first delete such bins
 - The only bin that summarizes tuples, some of which are older than W is T_{m-1} . Error in the count is due to these older tuples.
 - Expected error in count is therefore $T_{m-1}.C/2$
 - Largest number of 1s possible is $\sum_{j=0}^{m-1} T_j.C$
 - Smallest number of 1s possible is $\sum_{j=0}^{m-2} T_j.C + 1$

Error Analysis

- Maximum expected relative error occurs with the smallest number of possible 1s
- Hence relative
$$err \leq \frac{T_{m-1} \cdot C}{2 \left(1 + \sum_{j=0}^{m-2} T_j \cdot C \right)}$$
- Recall how we split bins (based on k).
 - We have bin sizes that are increasing, and powers of 2
 - There are at least k and at most $k+1$ bins of size 1
 - There are at least $k/2$ and at most $(k/2+1)$ bins of all other sizes

$$err \leq \frac{T_{m-1} \cdot C}{2 \left(1 + \sum_{j=0}^{m-2} T_j \cdot C \right)} \leq \frac{1}{k}$$

Error Analysis

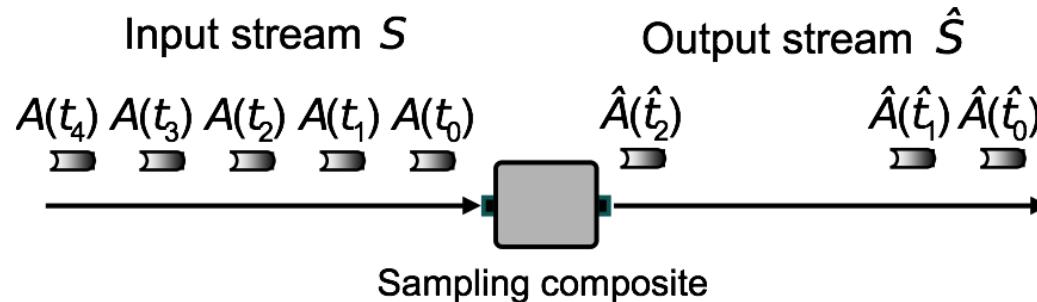
- Finally, since $k = \left\lceil \frac{1}{\varepsilon} \right\rceil$
- We have $err \leq \frac{T_{m-1} \cdot C}{2 \left(1 + \sum_{j=0}^{m-2} T_j \cdot C \right)} \leq \frac{1}{k} \leq \varepsilon$
- Relative error is bounded by ε
- Finally, we can show we only have $O\left(\frac{1}{\varepsilon} \log^2 W\right)$ bins
- BasicCounting algorithm extensions
 - From binary to produce counts of non-zero values if they lie in range $[0, R]$
 - Moving towards histogram estimation
- *Can you implement this algorithm?*

Basic Counting: Summary

- Need to tradeoff space and compute cost versus result accuracy in streaming scenario
- Design of a streaming algorithm non-trivial
 - Especially with bounds on error/space tradeoffs

Sampling

- Sampling is used to reduce the data stream by selecting some tuples based on one or more criteria



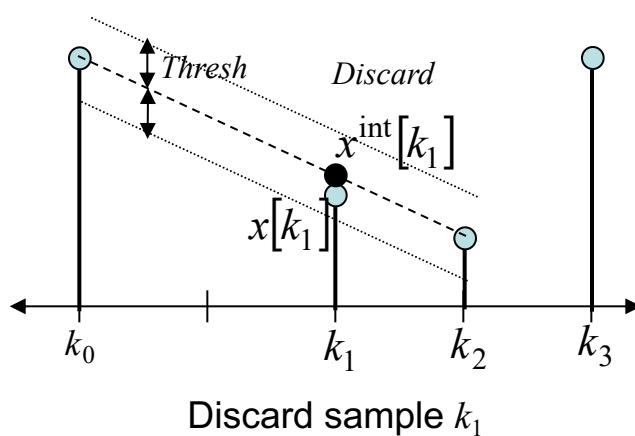
- Three types of sampling
 - Systematic or Uniform
 - Data Driven
 - Random

Systematic or Uniform Sampling

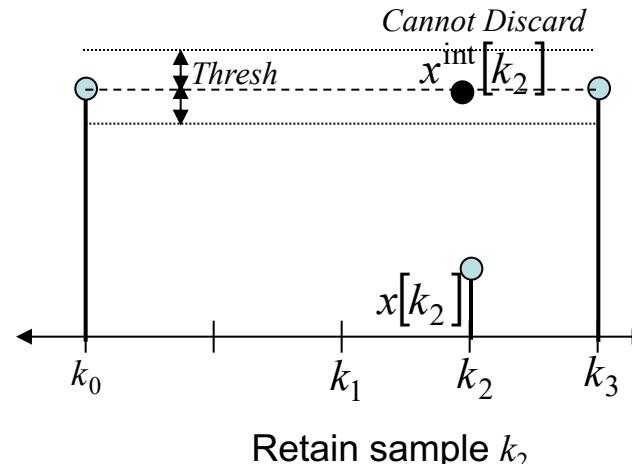
- Uniform Sampling
 - Sample every k -th tuple
 - Can be lossless
 - Recall Nyquist Sampling theorem and Aliasing
 - Simple streaming implementation
- Uniform sampling with random offset
 - Randomly pick a seed $[0, k-1]$
 - Starting from that sample every k -th sample
- Probability of sampling a tuple is $1/k$
- Number of retained samples grows with stream length
 - If we want to store samples – does this mean we have infinite samples when the data stream is infinite in length?

Data Driven Sampling

- Use data values to determine whether to sample or not
 - Does not result in uniformly spaced tuples even if original stream was uniformly spaced in time
 - Example: Interpolation driven sampling (SLIDE)
 - Based on prior values, or even future values



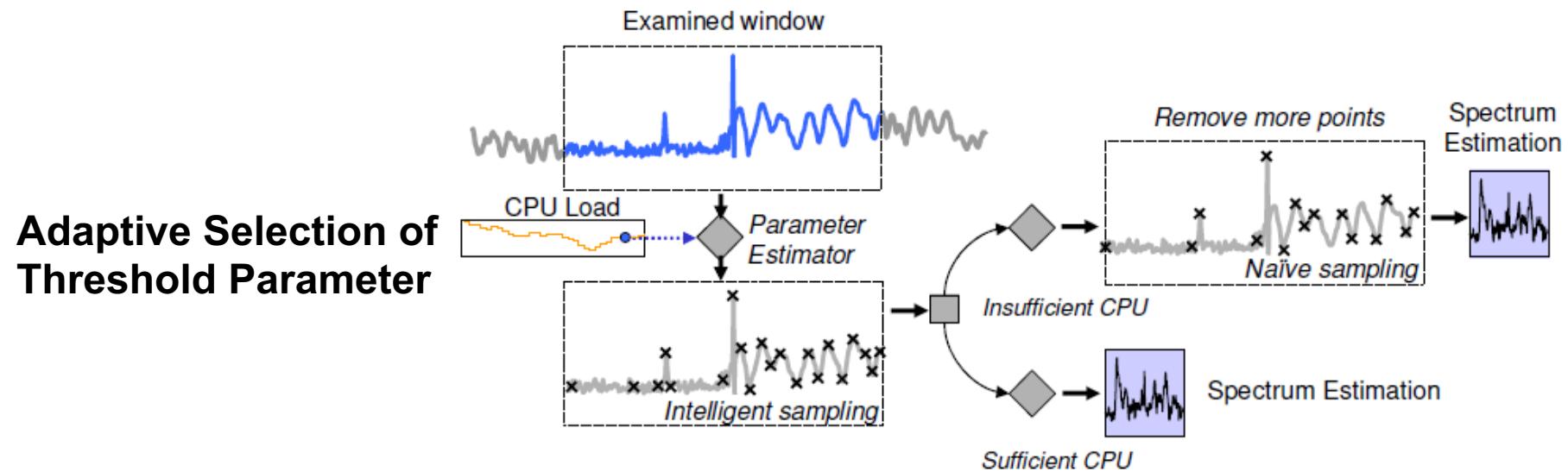
Discard sample k_1
Discard samples that can be predicted from neighbors



Retain sample k_2

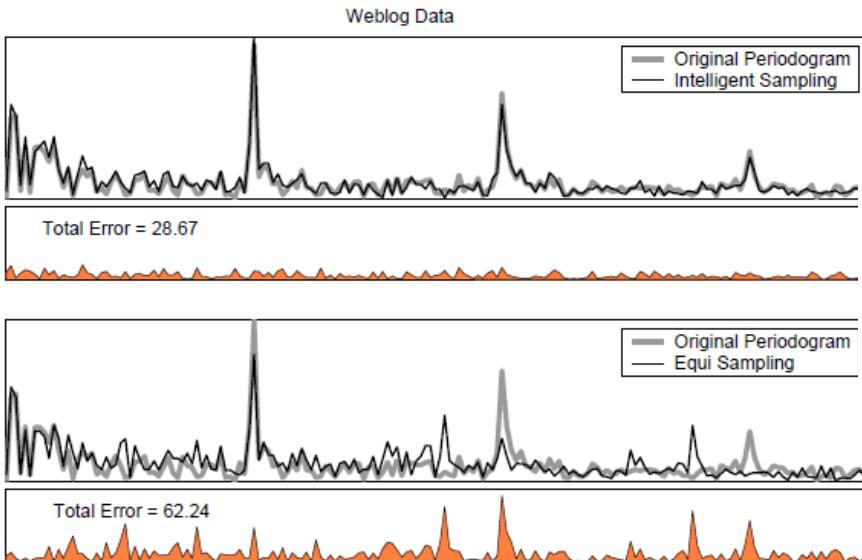
Data Driven Sampling

- Can result in much higher compression than uniform sampling (in terms of error recovery)
- What is the probability of sampling a tuple?
 - Dependent on tuple value. Unclear apriori
 - Depends on parameters, e.g. threshold. How do we set threshold?
 - Learn from previous window to set threshold for next window (can be done incrementally, requires storing of some features)



Data Driven Sampling

- Adds delay with look ahead
- Not trivial to estimate spectral properties of unevenly sampled signal
 - SLIDE allows some closed form estimates



Dataset	Threshold	Window Compression (%)	Error Equi-Sampling	Error Intelligent
ECG	20	80.96	1627.55	450.79
	60	91.40	2434.59	1326.23
	100	95.79	2934.84	2171.04
EEG	20	6.73	79.79	2.76
	60	18.45	202.03	33.10
	100	32.81	221.16	105.99
RTT	20	35.90	147.76	26.68
	60	60.69	174.24	81.21
	100	75.55	210.69	123.98
WebTrace	20	13.97	22.08	4.04
	60	37.26	46.29	18.98
	100	61.36	52.31	47.52

Random Sampling

- Select tuples randomly using some probability distribution
 - Used in statistics to get unbiased estimates
- Disadvantage of uniform and data-driven sampling (e.g. SLIDE)
 - Even if you sample, you still need infinite space, since the stream is endless
- Reservoir sampling
 - How do you maintain a reservoir of finite number of samples, such that every tuple has the same probability of being included
 - Independent of length of stream

Reservoir Sampling Algorithm

- Let q be the reservoir size
 - Indices in reservoir run from 0 to $q-1$
- Let i be the index of the tuple being processed right now
- If $i < q$
 - Append the tuple to the reservoir
- Otherwise
 - Select p as a random integer in range $[0, i]$
 - If $0 \leq p < q$
 - Replace tuple at index p in reservoir with current tuple

Analysis of Reservoir Sampling

- Consider that we have seen i tuples so far
- Consider a reservoir with size q
- Consider tuple k
 - We determine probability of this tuple being in the reservoir
- Case 1: $k < q$
 - This tuple was added to reservoir when it arrived
 - For this tuple to be in reservoir, no other tuple after q should have replaced it

$$p(k) = \prod_{j=q}^i \left(1 - \frac{1}{j+1}\right) = \frac{q}{i+1}$$

Analysis of Reservoir Sampling

- Case 2: $k \geq q$
 - This tuple can be added to the reservoir with probability

$$\frac{1}{k+1} / \frac{1}{q} = \frac{q}{k+1}$$

- If this was added, then probability it is not removed by subsequent tuples is

$$\prod_{j=k+1}^i \left(1 - \frac{1}{j+1}\right) = \frac{k+1}{i+1}$$

- Hence probability it stays in reservoir is

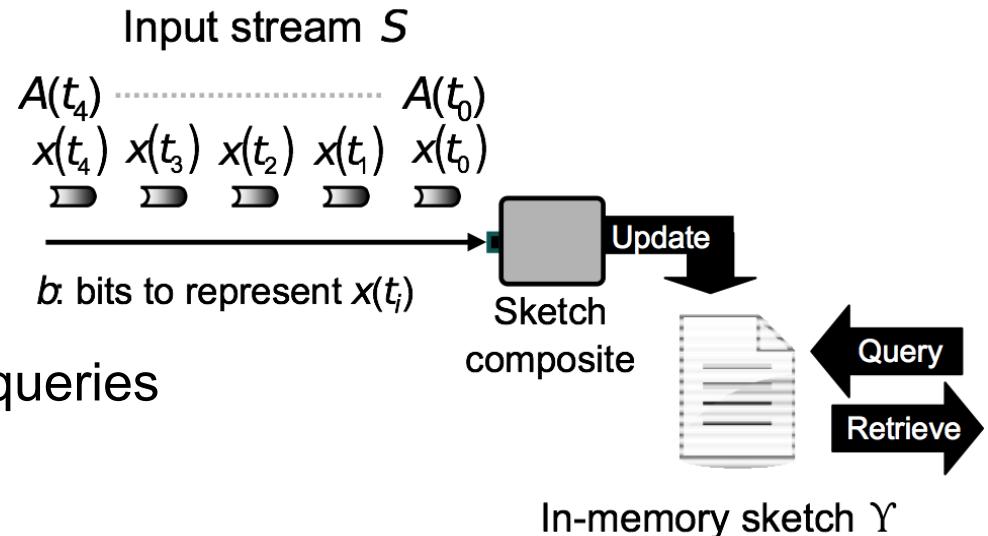
$$p(k) = \frac{q}{k+1} \frac{k+1}{i+1} = \frac{q}{i+1}$$

Analysis of Reservoir Sampling

- Probability of retaining any tuple
 - Only depends on reservoir size, and how many total tuples seen so far
 - Identical for all tuples: uniform sampling probability
- Originally proposed for tape deck compression
 - Pretty useful in streaming context
 - Recall: used during profiling
 - Easy to implement
- Reservoir sampling extended to sliding window case

Sketches

- In-memory data structures that contain compact, often lossy, synopses of streaming data
- They capture the key properties of the stream



- In order to answer specific queries
 - Error bounds and
 - probabilistic guarantees
- Used for approximate query processing

Popular Sketches

- Sketches to answer *frequency based queries*
 - Queries about the frequency distribution of a dataset
 - frequency of specific values
 - heavy hitters (frequently appearing values)
 - quantiles and equi-depth histograms
 - Examples: Count-Min sketch, AMS sketch
- Sketches to answer *distinct value based queries*
 - Queries about the cardinality of the stream
 - number of distinct values
 - Useful for cardinality estimation of relational queries
 - Examples: Flajolet-Martin, Gibbons-Tirthapura, BJKST

Aside on Hash Functions

```
function Hash(key)
    return key mod PrimeNumber
end

Additive Hash
ub4 additive(char *key, ub4 len, ub4 prime)
{
    ub4 hash, i;
    for (hash=len, i=0; i<len; ++i)
        hash += key[i];
    return (hash % prime);
}

Rotating Hash
ub4 rotating(char *key, ub4 len, ub4 prime)
{
    ub4 hash, i;
    for (hash=len, i=0; i<len; ++i)
        hash = (hash<<4)^(hash>>28)^key[i];
    return (hash % prime);
}
```

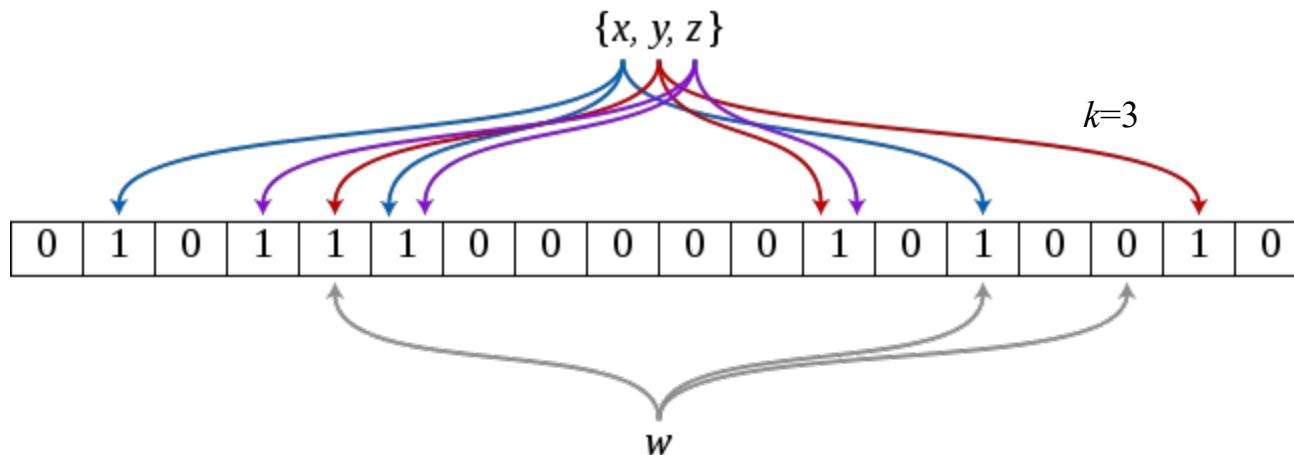
```
Bernstein's hash
ub4 bernstein(ub1 *key, ub4 len, ub4 level)
{
    ub4 hash = level;
    ub4 i;
    for (i=0; i<len; ++i) hash = 33*hash + key[i];
    return hash;
}
```

If your keys are lowercase English words, this will fit 6 characters into a 32-bit hash with no collisions

Several cryptographic hash functions, e.g. SHA2

Bloom Filters

- Let us say we want to answer containment queries
- Whether a given item has been seen so far or not
- Bloom filter
 - Keep a bit array of size m
 - Use k pairwise independent hash functions (more later)
 - Update: Hash item to k locations and set the bits to one
 - Query: Hash item to k locations
 - Return true if all of the k locations are set



Bloom Filters

- False negatives are not possible
 - If we say an item has not been seen, then it is not seen
- False positives are possible
 - If we say an item has been seen, we may be wrong
 - We want to set m, k to reach a desired possibility
- How to create pairwise independent hash functions?
 - In general difficult (we will discuss an example later)
 - Take a hash function with large range and slice into k parts
 - If a hash function takes a seed, give it k different seeds

Bloom Filter Analysis

- Probability of a bit not being set to 1 by a hash function (assuming hash function is uniform)

$$1 - \frac{1}{m}$$

- Probability of a bit not being set by k independent hash functions

$$\left(1 - \frac{1}{m}\right)^k$$

- Probability of a bit not being set after observing n values

$$\left(1 - \frac{1}{m}\right)^{kn}$$

- Hence probability of a bit being 1 after n values

$$1 - \left(1 - \frac{1}{m}\right)^{kn}$$

Bloom Filter Analysis

- Now, when we test for new value probability that all k bits are set to 1 is

$$\left[1 - \left(1 - \frac{1}{m}\right)^{kn}\right]^k$$

- This false positive probability may be approximated as

$$(1 - e^{-kn/m})^k$$

- Optimal number of hash functions for a given m and n is

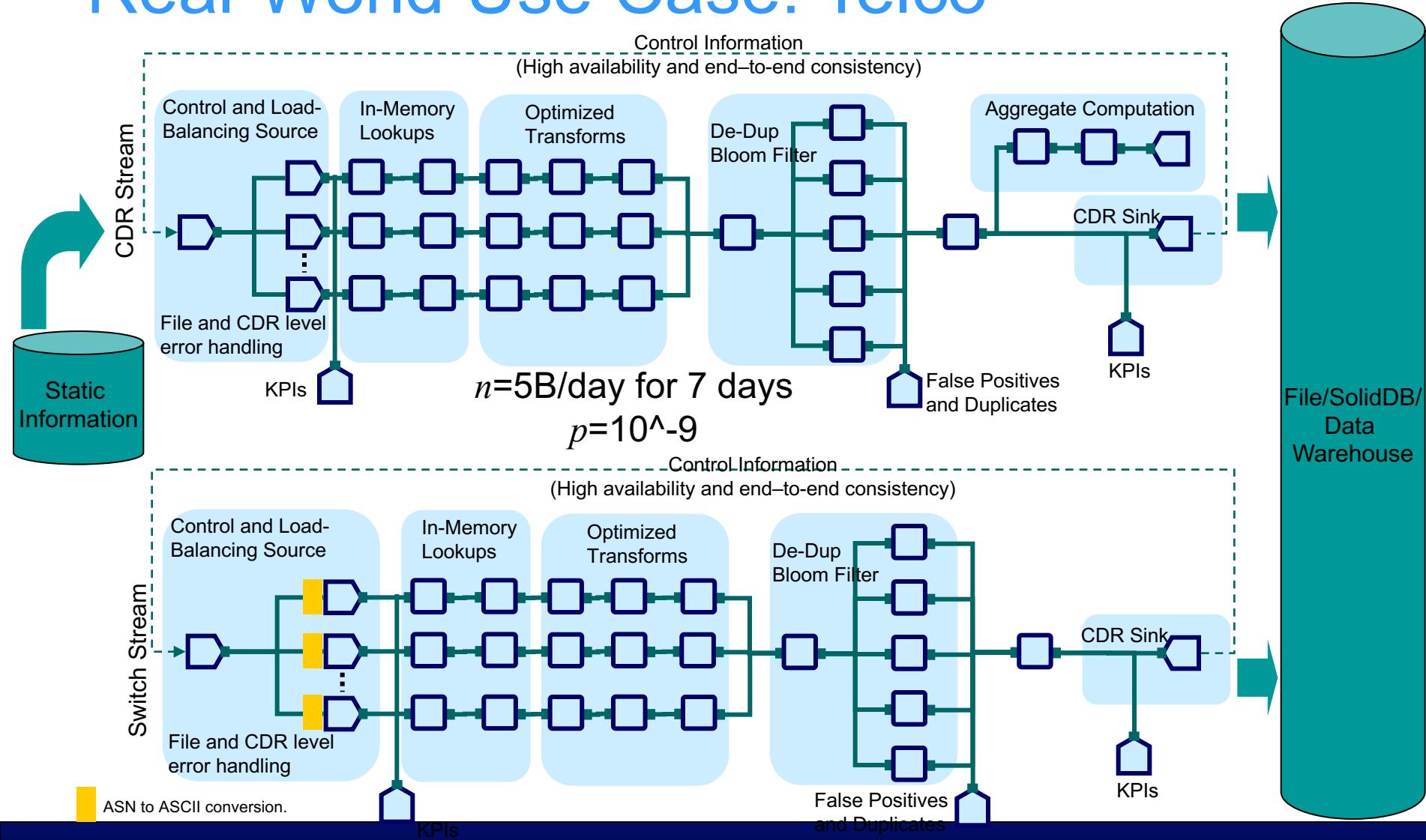
$$k = \frac{m}{n} \log 2$$

- Hence, given a n a desired false alarm rate p , we have

$$m = \frac{-n \ln p}{(\ln 2)^2}$$

Bloom filter calculator: <http://hur.st/bloomfilter?n=4&p=1.0E-20>

Real-World Use Case: Telco



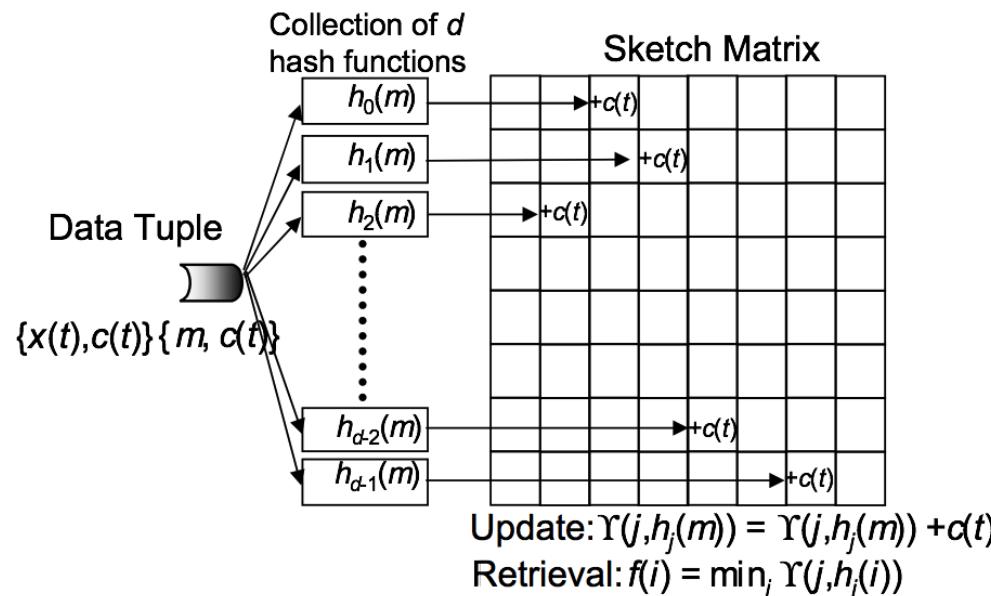
Count-Min Sketch

- Assume we have a numeric attribute with a very large domain
 - E.g. the domain of ip v6 addresses
- Assume we want to find, given a value, its frequency
- Let us use M to denote the domain size
- The Count-Min sketch keeps d rows and w columns
- Let H be a set of d pair-wise independent hash functions with range $[0..w)$

$$\Pr_{h \in H} [h(x_1) = y_1 \wedge h_k(x_2) = y_2] = \frac{1}{w^2}$$

Count-Min Sketch: Construction and Retrieval

- Construction: Apply the d hash functions
 - Increment the cell in each row whose index is the hash value
- Retrieval: Apply the d hash functions
 - Pick the smallest count value



Count-Min Sketch Properties

- Set $w = \lceil 2/\epsilon \rceil$ and $d = \lceil \log(1/\delta) \rceil$
- With probability $1 - \delta$
 - The error in the estimate ($|original-estimate|$) is less than ϵ times the sum of all frequencies
- Say δ is 0.01 and ϵ is 10^{-6}
- So with probability 0.99, our estimate is going to be within $\pm 10^{-6}$ of the original count
- This will require 20MB state
- 5 hash functions (very quick update and query)

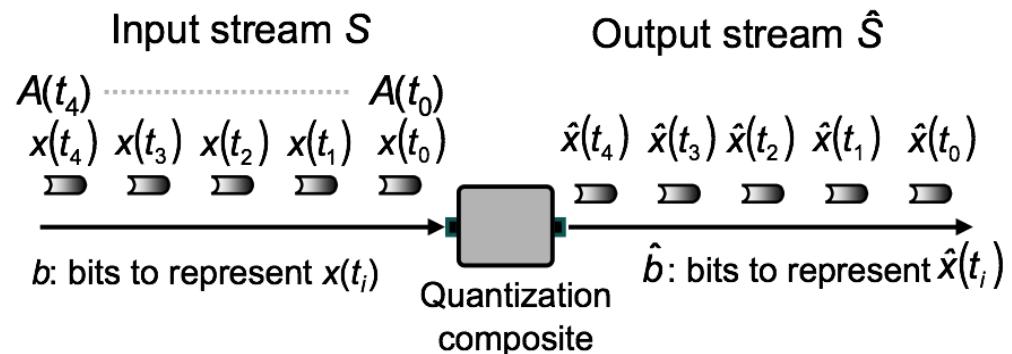
Generating Independent Hash Functions

- Pick a prime p in the range $w < p < 2w$
 - such a prime exists due to Bertrand-Chebyshev theorem
- For each j in $[0..d)$
 - Pick two random numbers $a_j > 0$ and b_j in range $[0..p)$
 - Set

$$h_j : x \rightarrow ((a_j \cdot x + b_j) \bmod p) \bmod w$$

Quantization

- Simplest lossy data reduction mechanism
 - Perfect reconstruction of original signal not possible
- Maps Input Sample → Reconstruction Value from predefined codebook
- Example
 - Rounding Function
 - Floor Function
 - Truncation
- Original sample
 - Numeric Tuples with Continuous or Discrete Space



Quantization

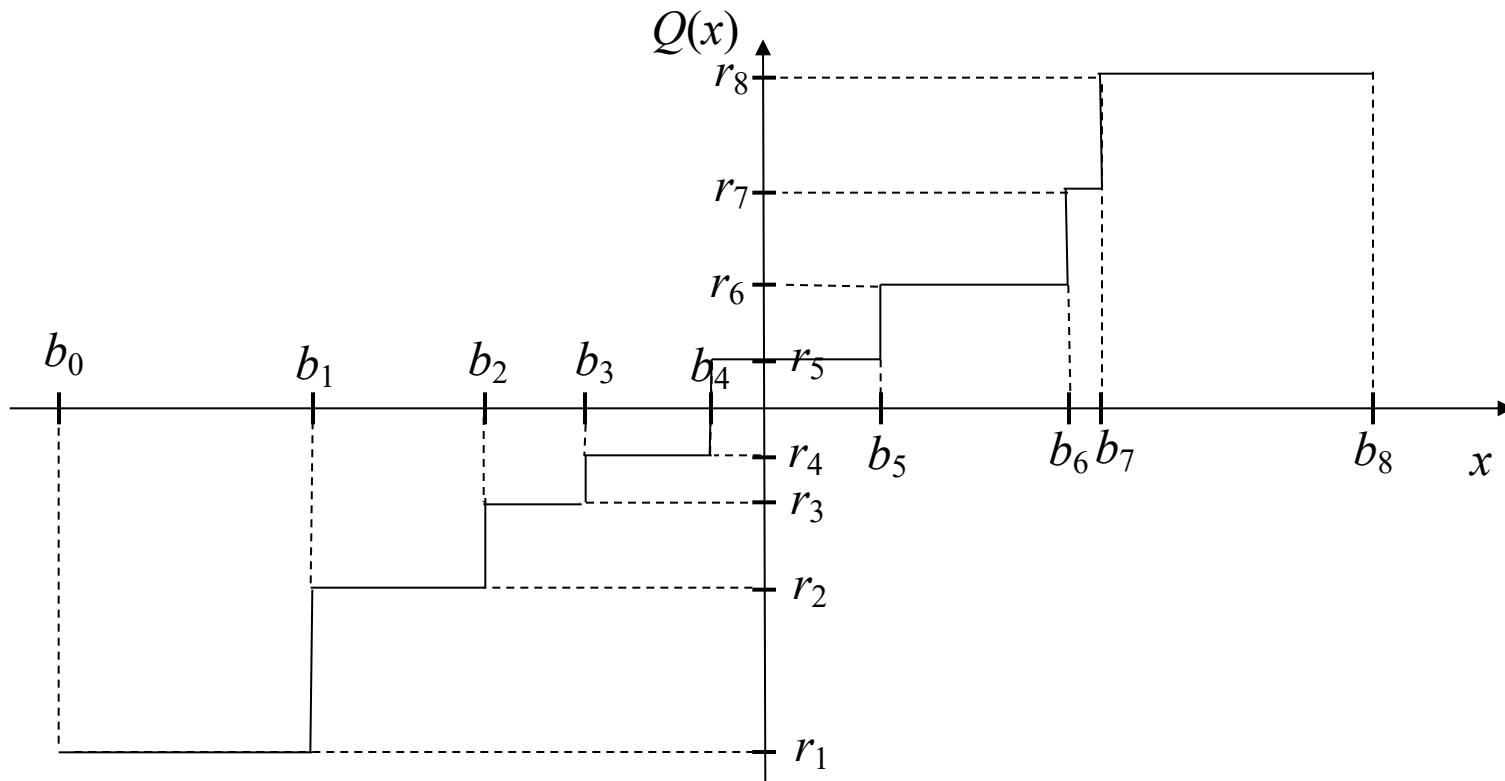
- We go from b bits to b' bits, where $b' < b$
- The *quantization error*: A function on the difference between the original and the quantized value
- Advantages:
 - Reduces memory requirement
 - Reduces computation requirements
- Two main techniques
 - Scalar quantization
 - apply on a single attribute
 - Vector quantization
 - apply on a vector of attributes

Scalar Quantization

- Quantizer Q described by
 - L : the number of reconstruction levels
 - Set $L = \{1, 2, \dots, L\}$ of reconstruction level indices
 - Requires at most $(\log_2 L)$ bits for representation
 - b_l : the set of boundary values
 - Boundary region $B_l = [b_{l-1}, b_l)$
 - Support space $B = \{B_l\}$
 - r_l : the set of reconstruction levels
 - $$Q(x) = r_l \text{ if } x \in B_l, l \in L$$

Partitions the support space of random variable x into L regions
Represents x with L discrete values

Scalar Quantization



Graphical Representation

Scalar Quantization

- Select boundaries and reconstruction levels to minimize reconstruction error
 - Mean absolute error
 - Mean squared error
- Multiple Greedy Optimization Techniques
 - Lloyd-Max algorithm (not a streaming algorithm)

Scalar Quantization

$$D_Q = E[d(X, Q(X))] = \int_{x \in} d(x, Q(x)) p_X(x) dx$$

↑
Distance function ↑
Probability density

$$d(x, y) = |x - y|$$

$$d(x, y) = |x - y|^2$$

Mean squared error (MSE)

$$p_X(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Gaussian

$$p_X(x) = \begin{cases} \frac{1}{b-a} & a \leq x < b \\ 0 & \text{otherwise} \end{cases}$$

Uniform

$$p_X(x) = \frac{1}{2b} e^{-\frac{|x-\mu|}{b}}$$

Laplacian

Scalar Quantization

$$D_Q = \int_{x \in} d(x, Q(x)) p_X(x) dx$$

$$D_Q = \sum_{l=1}^L \int_{b_{l-1}}^{b_l} d(x, r_l) p_X(x) dx$$

$$D_Q = \sum_{l=1}^L \int_{b_{l-1}}^{b_l} (x - r_l)^2 p_X(x) dx$$

MSE distance function

To derive the optimal or Minimum Mean Squared Error Quantizer, take derivatives w.r.t. r_l and b_l and set to zero

Scalar Quantization

$$\frac{\partial D_Q}{\partial r_l} = 0 \Rightarrow \int_{b_{l-1}}^{b_l} 2(x - r_l) p_X(x) dx = 0$$

$$r_l = \frac{\int_{b_{l-1}}^{b_l} x p_X(x) dx}{\int_{b_{l-1}}^{b_l} p_X(x) dx} = E[X | X \in B_l]$$

Optimal reconstruction values lie at the centroid of each boundary region

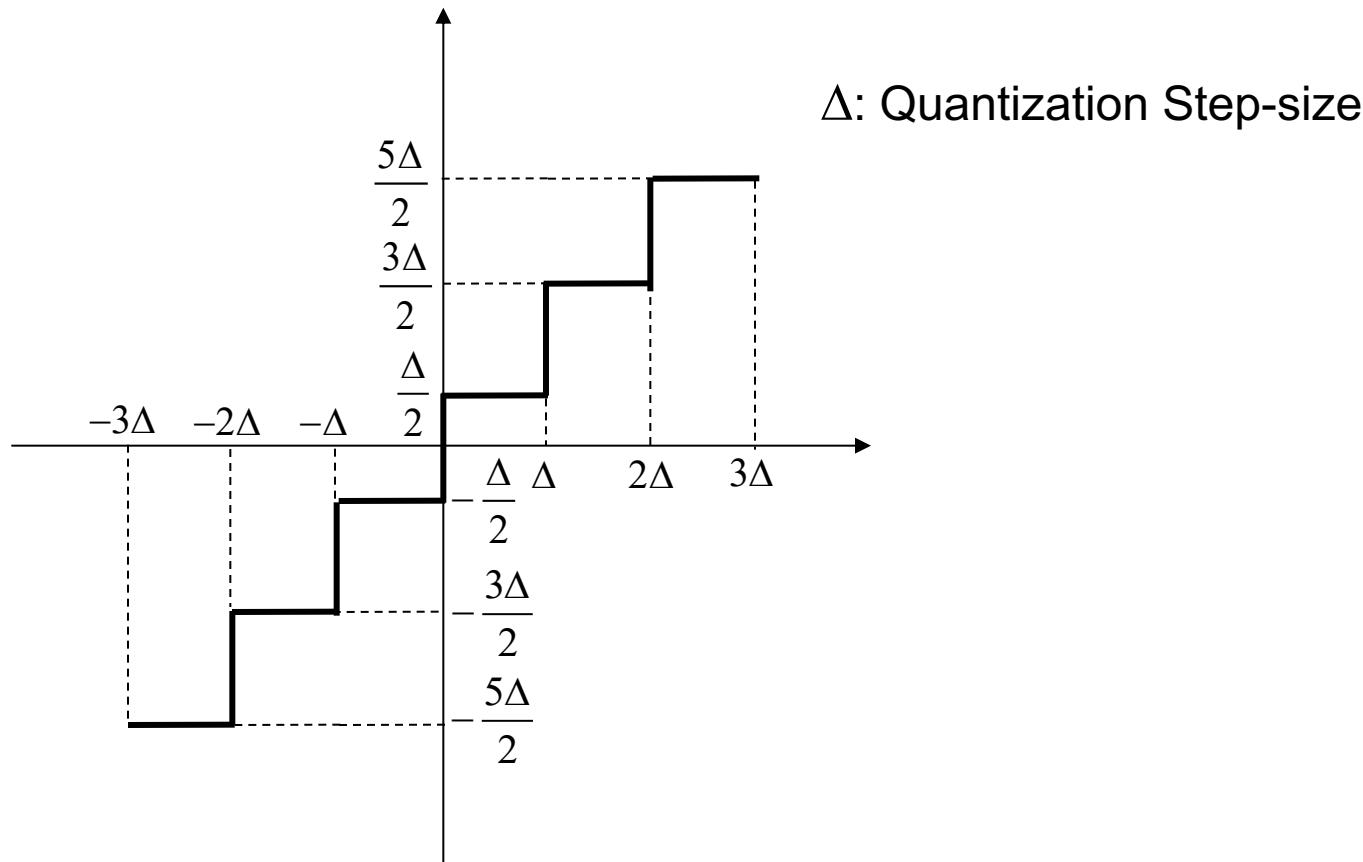
$$\frac{\partial D_Q}{\partial b_l} = 0 \Rightarrow (b_l - r_l)^2 p_X(b_l) - (b_l - r_{l+1})^2 p_X(b_l) = 0$$

$$b_l = \frac{r_{l+1} + r_l}{2}$$

Lloyd-Max Algorithm

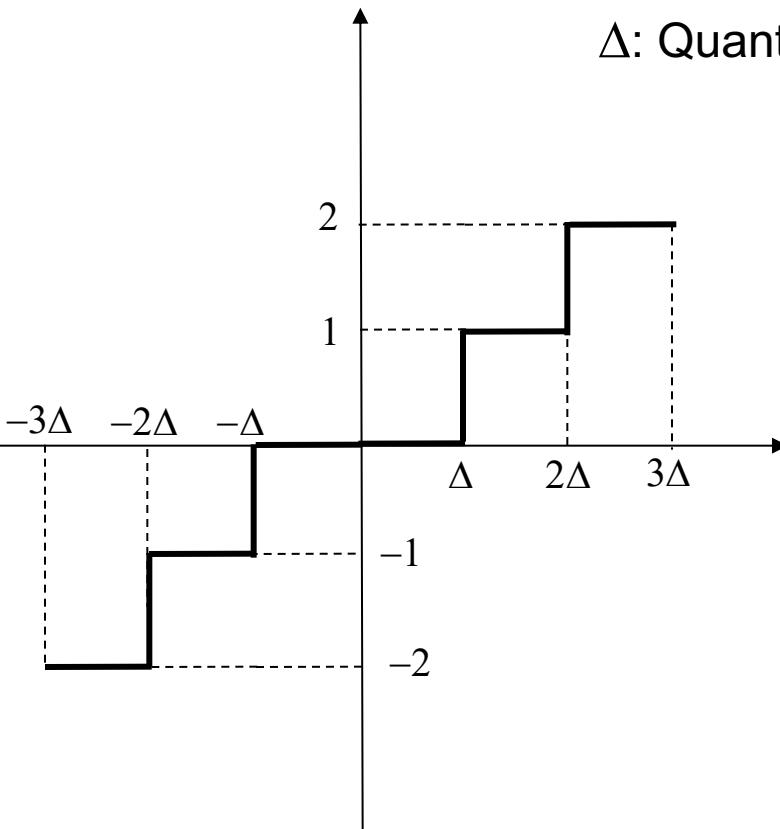
- Iterative Algorithm to determine r_l and b_l
 - Based on a set of training samples
- Step 1: Pick a set of reconstruction levels
- Step 2: Determine boundary partitions
 - Average of reconstruction levels (use nearest neighbor condition if discrete set of values)
- Step 3: Update reconstruction levels as centroids of each boundary region
- Step 4: Compute Quantization Distortion
- Iterate Steps 2 through 4 till Distortion converges
- Gradient Descent Based Algorithm
 - Convergence guaranteed to only Local Minimum (not necessarily Global Minimum)
- *Is this Streaming?*

Uniform Quantizer



Optimal MMSE Quantizer
for uniform distribution

Uniform Quantizer: Mid-tread



Uniform Mid-Tread Quantizer
Quantization with “Dead Zone”

Δ : Quantization Step-size

“Dead Zone”

$$Q(x) = sign(x) \times \frac{|x + f(\Delta)|}{\Delta}$$

$\hat{x} = \Delta Q(x)$

Reconstruction of original

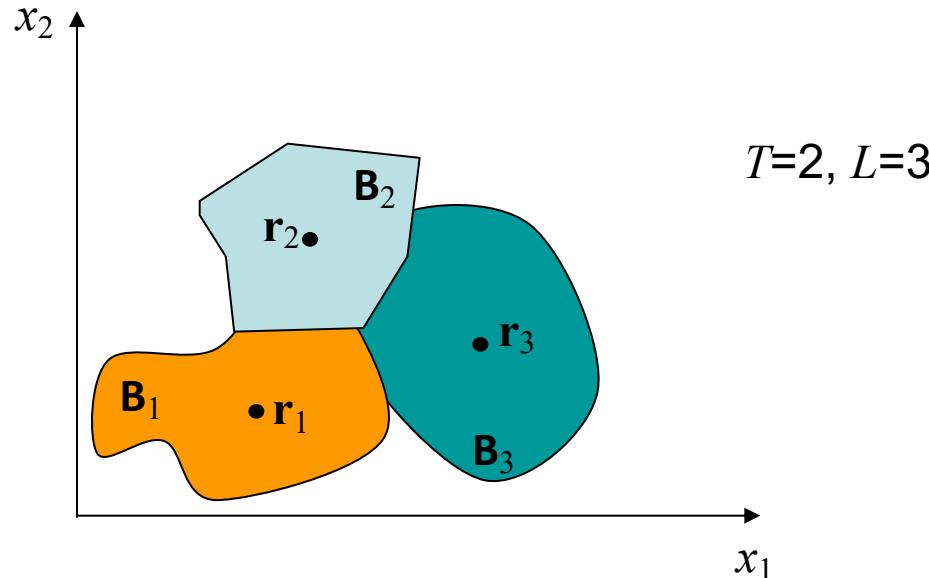
Vector Quantization

- Quantizer Q described by
 - T : the number of dimensions per vector
 - L : the number of reconstruction vectors
 - Set $\mathcal{L} = \{1, 2, \dots, L\}$ of reconstruction vector indices
 - Boundary region B_l
 - Support space $\mathbf{B} = \{B_l\}$
 - \mathbf{r}_l : the set of reconstruction vectors (codewords)

$$Q(\mathbf{x}) = \mathbf{r}_l \quad \text{if } \mathbf{x} \in B_l, \quad l \in \mathcal{L}$$

Partitions the support space of random vector \mathbf{x} into L regions
Represents \mathbf{x} with one of L discrete values

Illustration: 2D Vector Quantization



Represents vectors in this 2-D space with 3 reconstruction vectors

Bit-rate required after quantization $\sim \log_2 3$ per tuple $\sim 1/T(\log_2 3)$ per dimension

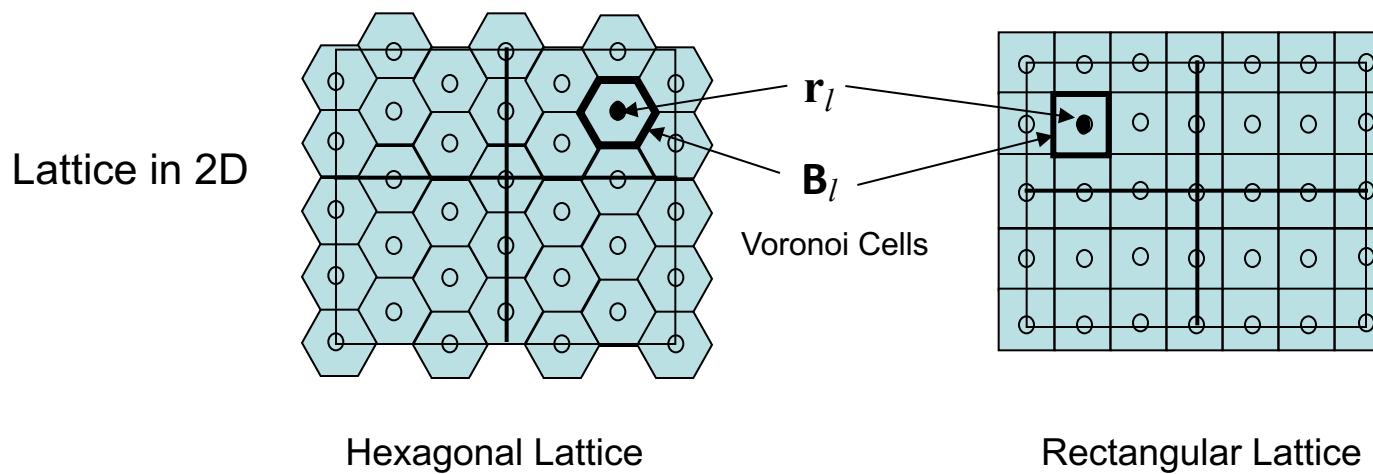
Quantization – analogous to clustering: Similar goals and algorithms

Vector Quantization Algorithms

- Linde Buzo and Gray (LBG) Algorithm
 - Analogous to the Lloyd-Max algorithm
- Choice of initial codebook
 - A representative subset of the training vectors
 - e.g., some face blocks, some shirt blocks, some background blocks
 - Scalar quantization in each dimension
 - Splitting...
- Nearest Neighbor (NN) algorithm [Equitz, 1984]
 - Start with the entire training set
 - Merge the two vectors that are closest into one vector equal to their mean
 - Repeat until the desired number of vectors is reached, or the distortion exceeds a certain threshold

Uniform Vector Quantizer

Uniform vector quantizer is a lattice quantizer



Recall: Analogy to sampling

Codebook contains all lattice point (reconstruction vectors)
Nearest neighbor property used for quantization

Streaming Algorithms

Objectives

- Seminar
- Streaming Algorithms for Pre-processing
 - Recap
 - Descriptive Statistics, Sampling, Sketches
 - Quantization
 - Transforms
 - Dimensionality Reduction

Principles

- Avoid multi-pass algorithms
 - Data available streaming
- Need efficiency in space-compute-complexity
 - Cannot keep large amounts in memory
- Approximate answers are acceptable
 - Bounds on approximation error desirable

Principles

- Support distributed implementations on SPS
 - E.g. split using data parallelism
 - Stateful versus stateless processing
- Tradeoff data versus process time
 - Understand interaction with sliding/tumbling windows

Principles: Approximation Error

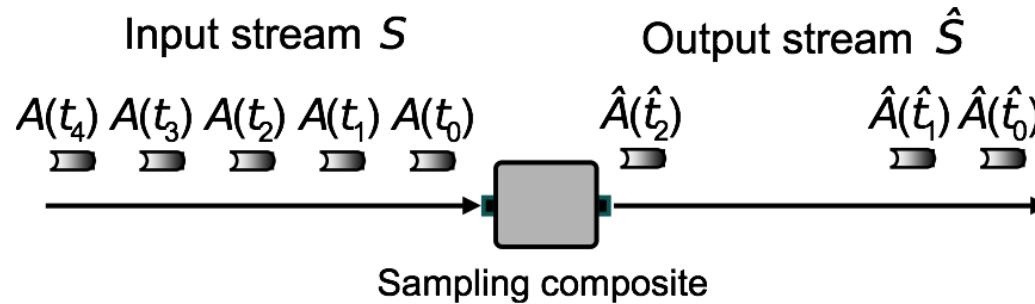
- Depend on the nature of the aggregation
 - Counts/Histogram
 - Additive Aggregations
 - Multi-input Data
- Bounds on Error
 - No Bounds
 - Deterministic Bounds
 - Probabilistic Bounds

BasicCounting Algorithm

- Question: Assume you have a stream that contains tuples that are either 0s or 1s, how can you maintain the number of 1s in the last W tuples
 - Sliding window of size W , slide 1
- Assume W is large
 - So $O(W)$ is too large. We want to store logarithmic space
- The *BasicCounting* algorithm
 - Uses space of the order $O\left(\frac{1}{\varepsilon} \log^2 W\right)$
 - With error tolerance ε , i.e. the counts are within $1 \pm \varepsilon$ of its actual value. ε is specified by application objective
- Study: Data structure, update, query

Sampling

- Sampling is used to reduce the data stream by selecting some tuples based on one or more criteria



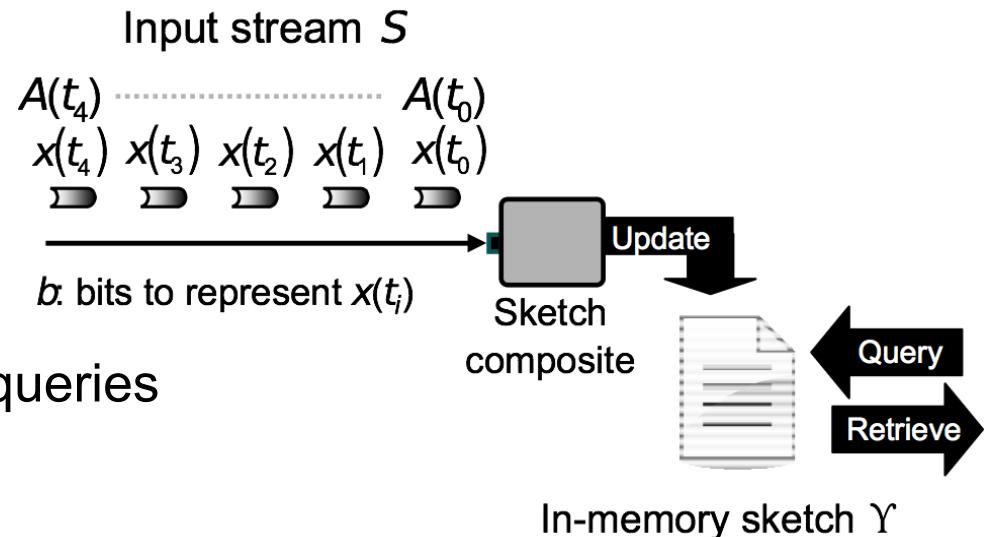
- Three types of sampling
 - Systematic or Uniform
 - Data Driven
 - Random

Summary

Techniques	Tuple Types	Windowing
Moments	Numeric Tuple, Univariate or Multivariate	Sliding or tumbling window
Counts and Histograms (BasicCounting)	Numeric or non-numeric data	Sliding or tumbling window
Systematic Sampling	Numeric or non-numeric data	Not windowed. Can be tumbling window
Data-driven Sampling	Numeric or non-numeric data (Interpolation driven is for numeric data)	Not windowed. Can be tumbling window
Random Sampling	Numeric or non-numeric data	Not windowed. Can be tumbling window

Sketches

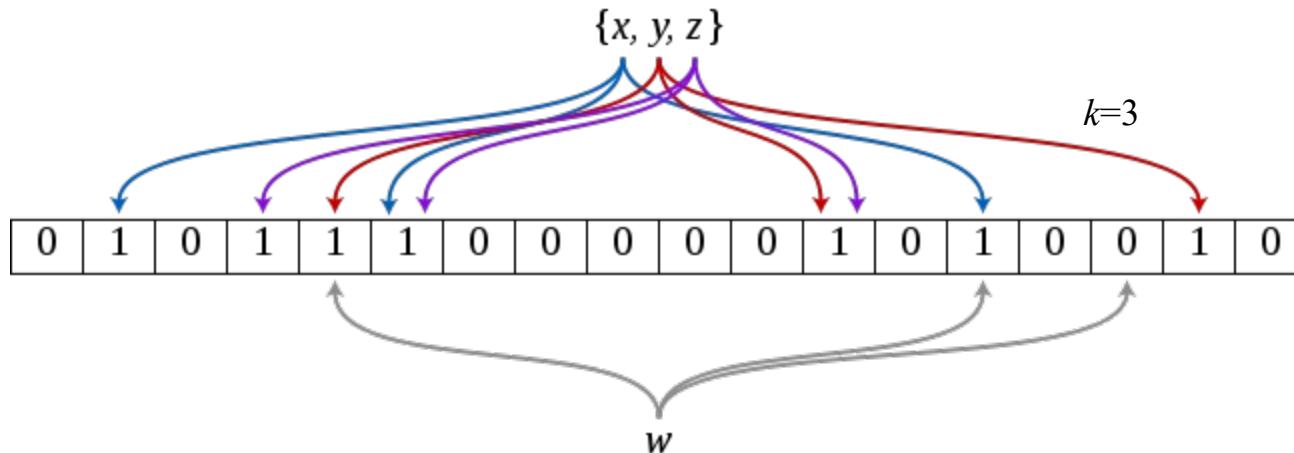
- In-memory data structures that contain compact, often lossy, synopses of streaming data
- They capture the key properties of the stream



- In order to answer specific queries
 - Error bounds and
 - probabilistic guarantees
- Used for approximate query processing

Bloom Filters

- Let us say we want to answer containment queries
- Whether a given item has been seen so far or not
- Bloom filter
 - Keep a bit array of size m
 - Use k pairwise independent hash functions (more later)
 - Update: Hash item to k locations and set the bits to one
 - Query: Hash item to k locations
 - Return true if all of the k locations are set



Bloom Filters

- False negatives are not possible
 - If we say an item has not been seen, then it is not seen
- False positives are possible
 - If we say an item has been seen, we may be wrong
 - We want to set m, k to reach a desired possibility
- How to create pairwise independent hash functions?
 - In general difficult (we will discuss an example later)
 - Take a hash function with large range and slice into k parts
 - If a hash function takes a seed, give it k different seeds

Generating Independent Hash Functions

- Pick a prime p in the range $m < p < 2m$
 - such a prime exists due to Bertrand-Chebyshev theorem
- For each j in $[0..k)$
 - Pick two random numbers $a_j > 0$ and b_j in range $[0..p)$
 - Set

$$h_j : x \rightarrow ((a_j x + b_j) \bmod p) \bmod m$$

$$\Pr [(h_j(x_1) = y_1) \wedge (h_j(x_2) = y_2)] = 1/m^2$$

Bloom Filter Analysis

- Probability of a bit not being set to 1 by a hash function (assuming hash function is uniform)

$$1 - \frac{1}{m}$$

- Probability of a bit not being set by k independent hash functions

$$\left(1 - \frac{1}{m}\right)^k$$

- Probability of a bit not being set after observing n values

$$\left(1 - \frac{1}{m}\right)^{kn}$$

- Hence probability of a bit being 1 after n values

$$1 - \left(1 - \frac{1}{m}\right)^{kn}$$

Bloom Filter Analysis

- Now, when we test for new value probability that all k bits are set to 1 is

$$\left[1 - \left(1 - \frac{1}{m}\right)^{kn}\right]^k$$

- This false positive probability may be approximated as

$$(1 - e^{-kn/m})^k$$

- Optimal number of hash functions for a given m and n is

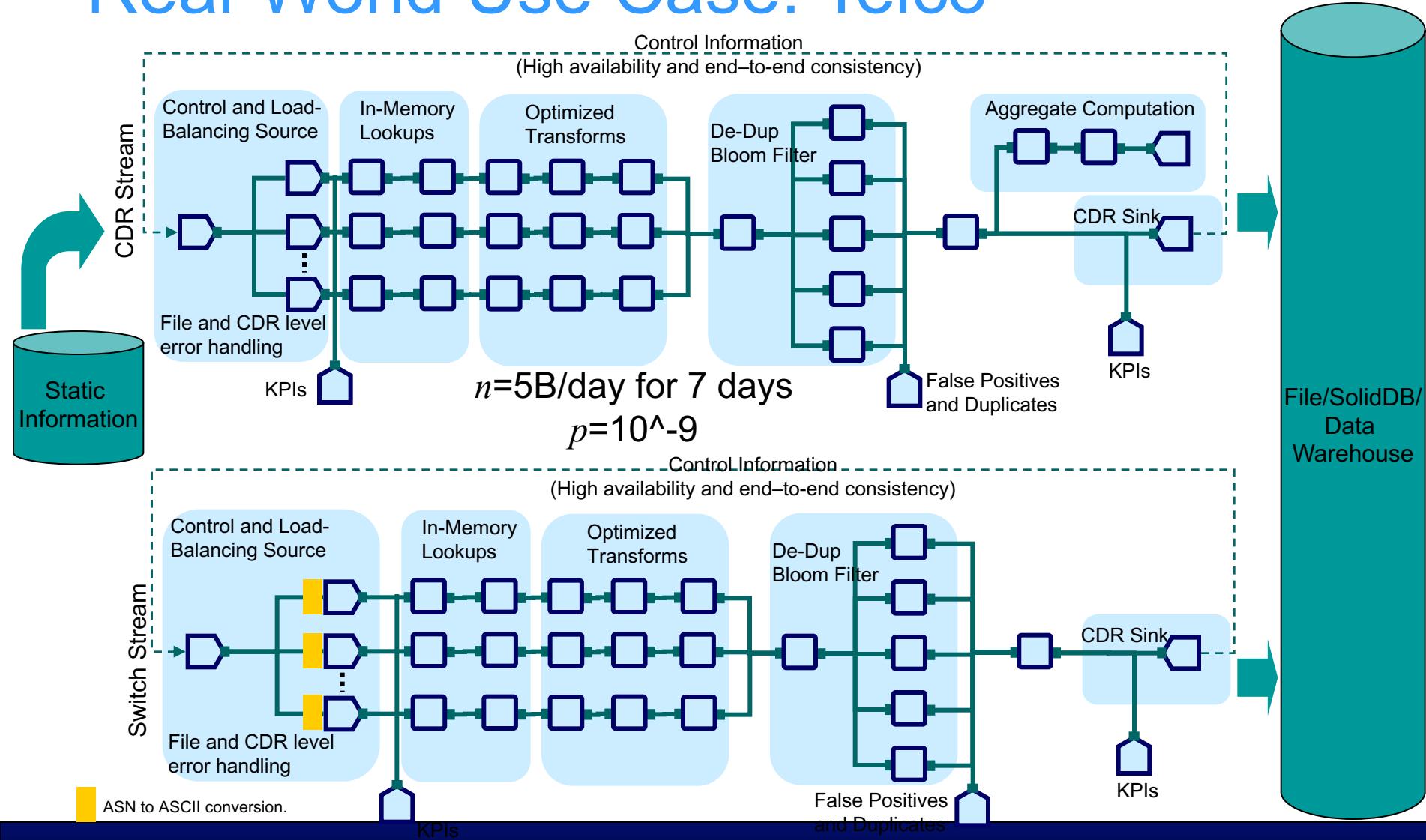
$$k = \frac{m}{n} \log 2$$

- Hence, given a n a desired false alarm rate p , we have

$$m = \frac{-n \ln p}{(\ln 2)^2}$$

Bloom filter calculator: <http://hur.st/bloomfilter?n=4&p=1.0E-20>

Real-World Use Case: Telco

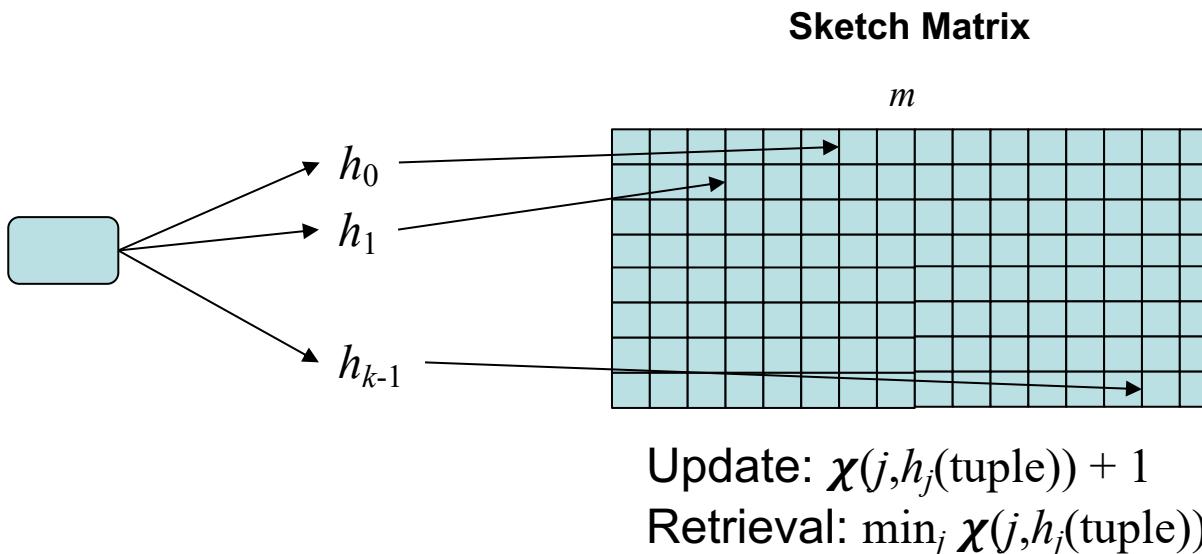


Count-Min Sketch

- Assume we have a numeric attribute with a very large domain
 - E.g. the domain of ip v6 addresses
- Assume we want to find, given a value, its frequency
- Let us use D to denote the domain size
- The Count-Min sketch keeps k rows and m columns
- Let H be a set of k pair-wise independent hash functions with range $[0..m)$

Count-Min Sketch: Construction and Retrieval

- Construction: Apply the k hash functions
 - Increment the cell in each row whose index is the hash value
- Retrieval: Apply the k hash functions
 - Pick the smallest count value

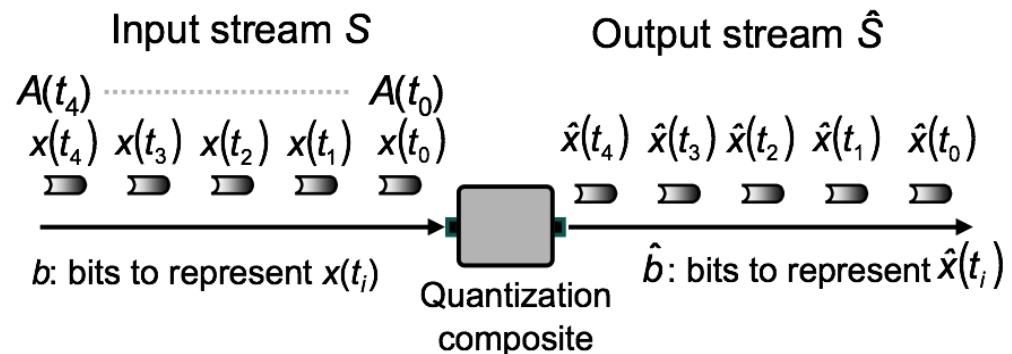


Count-Min Sketch Properties

- Set $m = \lceil 2/\varepsilon \rceil$ and $k = \lceil \log(1/\delta) \rceil$
- With probability $1 - \delta$
 - The error in the estimate ($|\text{original}-\text{estimate}|$) is less than ε times the sum of all frequencies
- Say δ is 0.01 and ε is 10^{-6}
- So with probability 0.99, our estimate is going to be within $\pm 10^{-6}$ of the original count
- This will require 20MB state
- 5 hash functions (very quick update and query)

Quantization

- Simplest lossy data reduction mechanism
 - Perfect reconstruction of original signal not possible
- Maps Input Sample → Reconstruction Value from predefined codebook
- Example
 - Rounding Function
 - Floor Function
 - Truncation
- Original sample
 - Numeric Tuples with Continuous or Discrete Space



Quantization

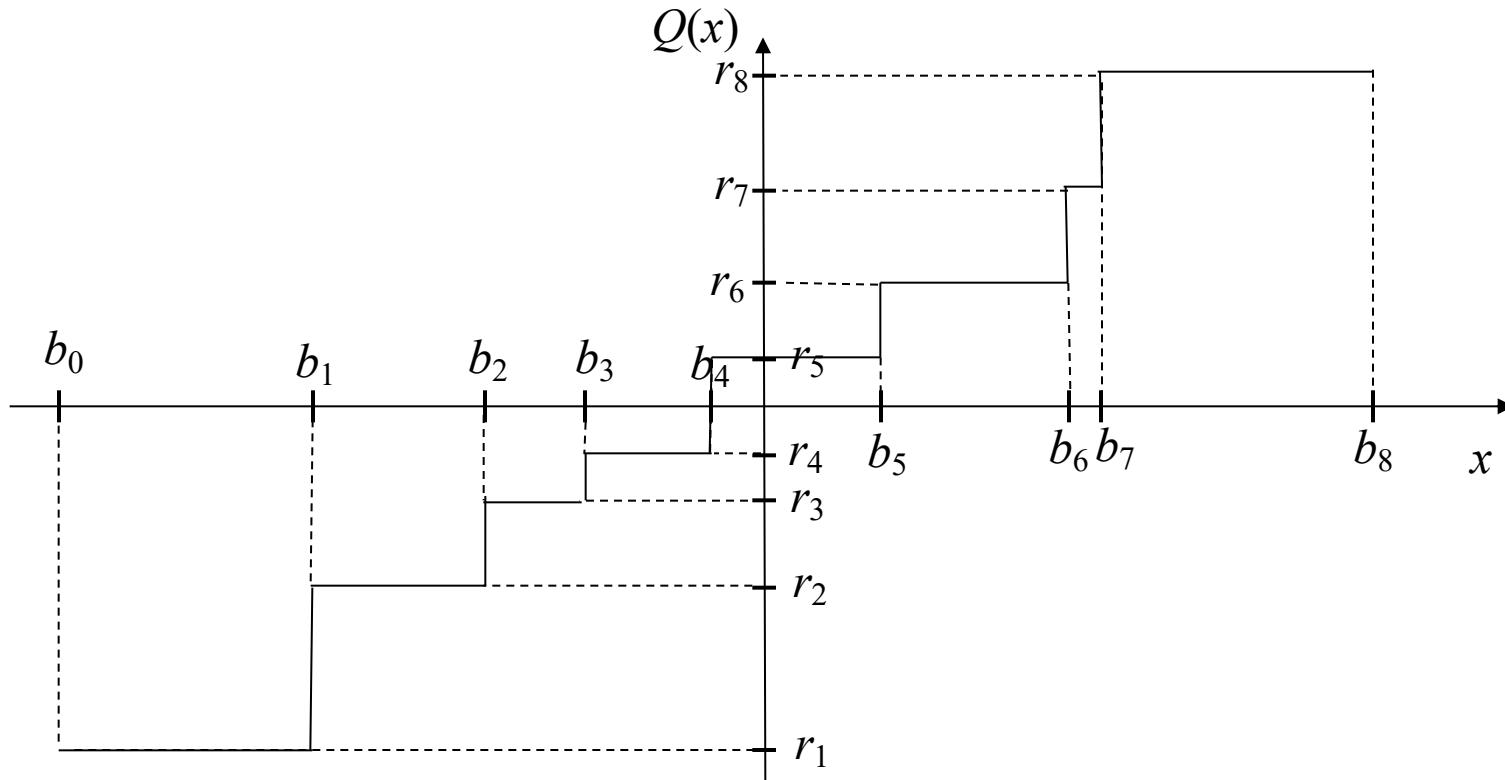
- We go from b bits to b' bits, where $b' < b$
- The *quantization error*: A function on the difference between the original and the quantized value
- Advantages:
 - Reduces memory requirement
 - Reduces computation requirements
- Two main techniques
 - Scalar quantization
 - apply on a single attribute
 - Vector quantization
 - apply on a vector of attributes

Scalar Quantization

- Quantizer Q described by
 - L : the number of reconstruction levels
 - Set $\mathcal{L}=\{1,2,\dots,L\}$ of reconstruction level indices
 - Requires at most $(\log_2 L)$ bits for representation
 - b_l : the set of boundary values
 - Boundary region $B_l = [b_{l-1}, b_l)$
 - Support space $B = \{B_l\}$
 - r_l : the set of reconstruction levels
 - $$Q(x) = r_l \text{ if } x \in B_l, l \in \mathcal{L}$$

Partitions the support space of random variable x into L regions
Represents x with L discrete values

Scalar Quantization



Graphical Representation

Quantizing Streaming Data

- Applying a known quantizer (codebook)
 - Stateless operation
 - Very efficient
 - Does not matter whether data is windowed or not
- If codebook shared downstream
 - Can transmit just index – higher compression
 - Applied to batches and combined with entropy coding (e.g. Huffman Coding/Arithmetic Coding)

Quantizer Design: Scalar Quantization

- Select boundaries and reconstruction levels to minimize reconstruction error
 - Mean absolute error
 - Mean squared error
- Multiple Greedy Optimization Techniques
 - Lloyd-Max algorithm (not a streaming algorithm)

Scalar Quantization

$$D_Q = E[d(X, Q(X))] = \int_{x \in} d(x, Q(x)) p_X(x) dx$$

↑
Distance function ↑
Probability density

$$d(x, y) = |x - y|$$

$$d(x, y) = |x - y|^2$$

Mean squared error (MSE)

$$p_X(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Gaussian

$$p_X(x) = \begin{cases} \frac{1}{b-a} & a \leq x < b \\ 0 & \text{otherwise} \end{cases}$$

Uniform

$$p_X(x) = \frac{1}{2b} e^{-\frac{|x-\mu|}{b}}$$

Laplacian

Scalar Quantization

$$D_Q = \int_{x \in} d(x, Q(x)) p_X(x) dx$$

$$D_Q = \sum_{l=1}^L \int_{b_{l-1}}^{b_l} d(x, r_l) p_X(x) dx$$

$$D_Q = \sum_{l=1}^L \int_{b_{l-1}}^{b_l} (x - r_l)^2 p_X(x) dx$$

MSE distance function

To derive the optimal or Minimum Mean Squared Error Quantizer, take derivatives w.r.t. r_l and b_l and set to zero

Scalar Quantization

$$\frac{\partial D_Q}{\partial r_l} = 0 \Rightarrow \int_{b_{l-1}}^{b_l} 2(x - r_l) p_X(x) dx = 0$$

$$r_l = \frac{\int_{b_{l-1}}^{b_l} x p_X(x) dx}{\int_{b_{l-1}}^{b_l} p_X(x) dx} = E[X | X \in B_l]$$

Optimal reconstruction values lie at the centroid of each boundary region

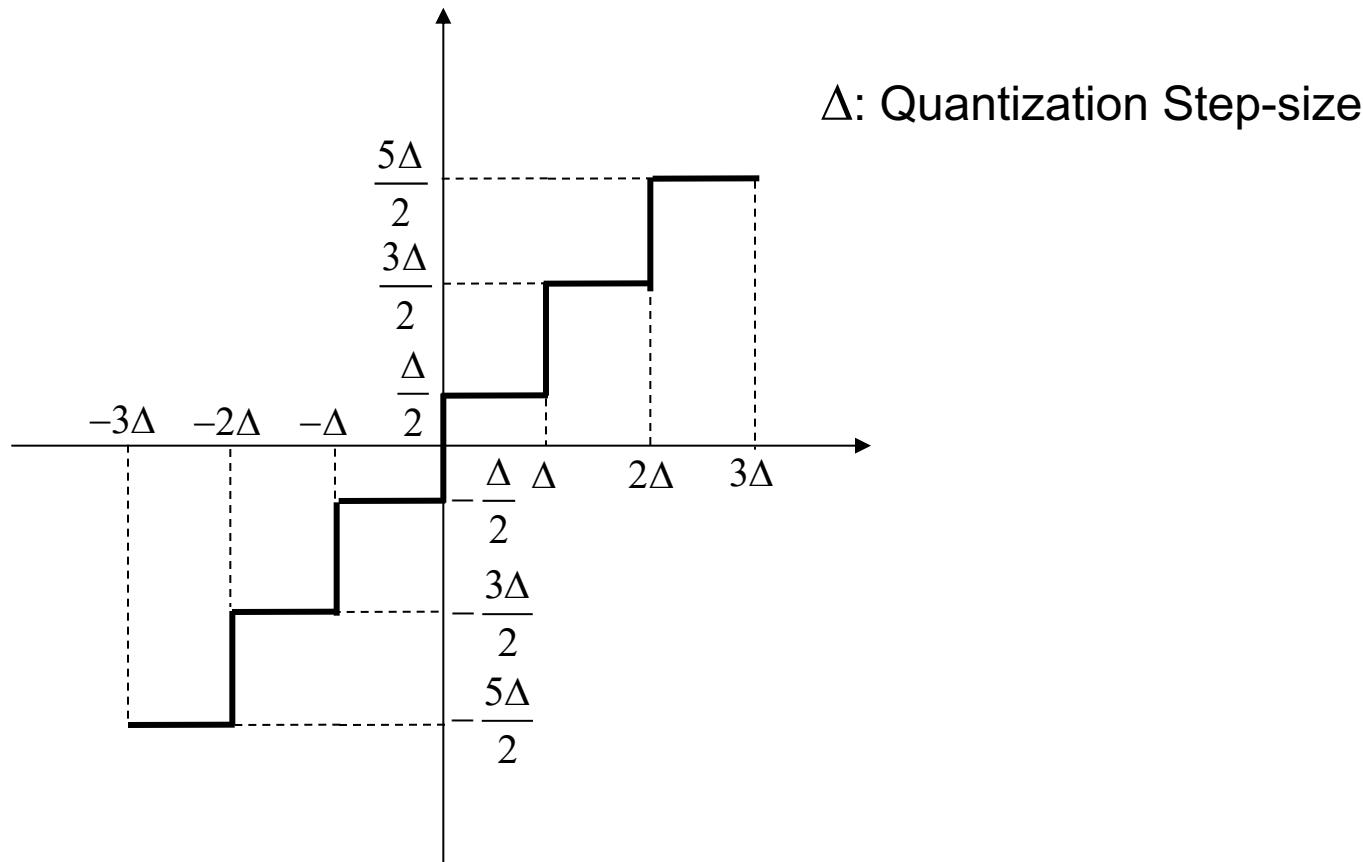
$$\frac{\partial D_Q}{\partial b_l} = 0 \Rightarrow (b_l - r_l)^2 p_X(b_l) - (b_l - r_{l+1})^2 p_X(b_l) = 0$$

$$b_l = \frac{r_{l+1} + r_l}{2}$$

Lloyd-Max Algorithm

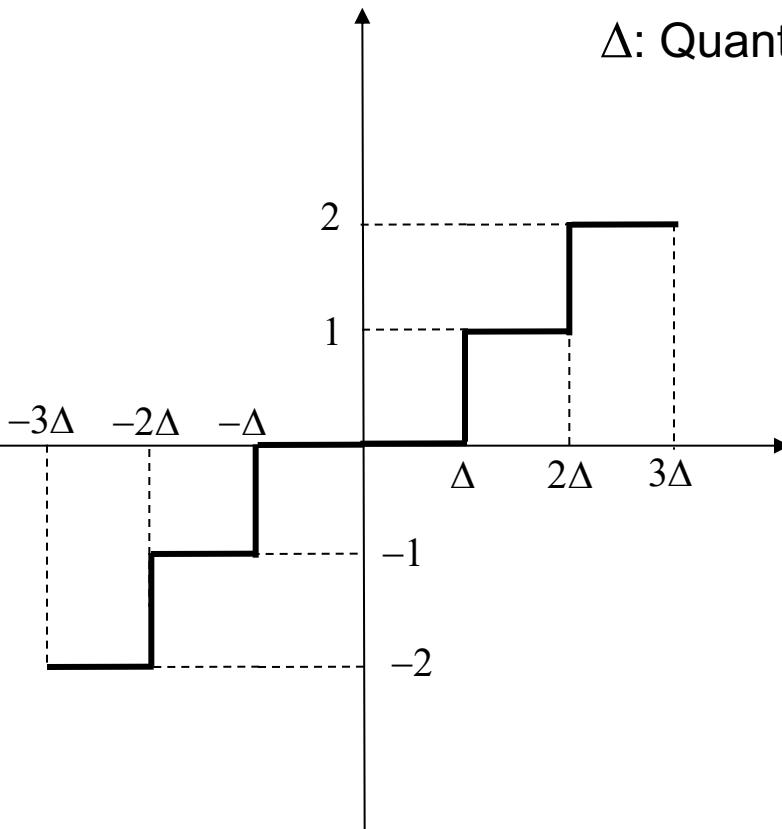
- Iterative Algorithm to determine r_l and b_l
 - Based on a set of training samples
- Step 1: Pick a set of reconstruction levels
- Step 2: Determine boundary partitions
 - Average of reconstruction levels (use nearest neighbor condition if discrete set of values)
- Step 3: Update reconstruction levels as centroids of each boundary region
- Step 4: Compute Quantization Distortion
- Iterate Steps 2 through 4 till Distortion converges
- Gradient Descent Based Algorithm
 - Convergence guaranteed to only Local Minimum (not necessarily Global Minimum)
- *Is this Streaming friendly?*

Uniform Quantizer



Optimal MMSE Quantizer
for uniform distribution

Uniform Quantizer: Mid-tread



Uniform Mid-Tread Quantizer
Quantization with “Dead Zone”

Δ : Quantization Step-size

$$Q(x) = sign(x) \times \frac{|x + f(\Delta)|}{\Delta}$$

$$\hat{x} = \Delta Q(x)$$

“Dead Zone”

Reconstruction of original

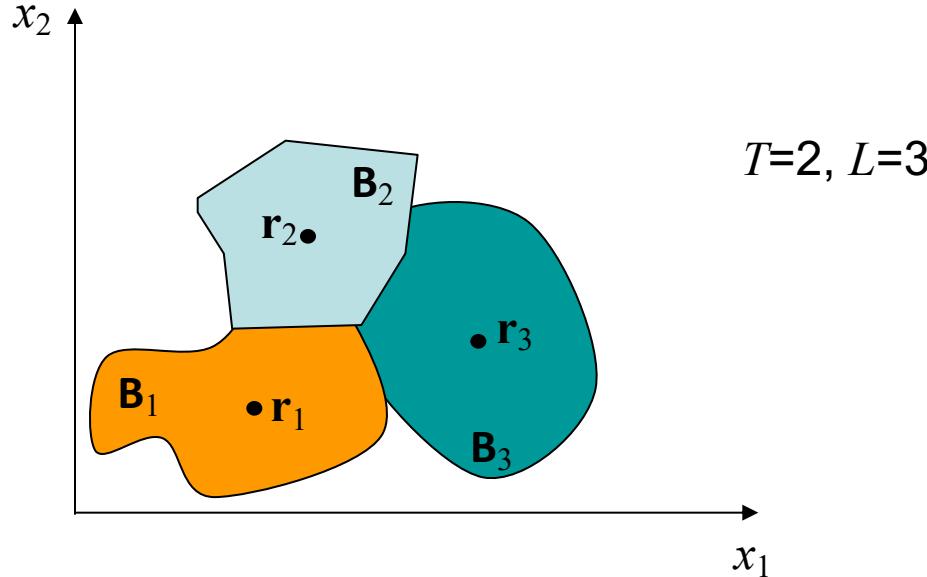
Vector Quantization

- Quantizer Q described by
 - T : the number of dimensions per vector
 - L : the number of reconstruction vectors
 - Set $\mathcal{L} = \{1, 2, \dots, L\}$ of reconstruction vector indices
 - Boundary region B_l
 - Support space $\mathbf{B} = \{B_l\}$
 - \mathbf{r}_l : the set of reconstruction vectors (codewords)

$$Q(\mathbf{x}) = \mathbf{r}_l \text{ if } \mathbf{x} \text{ in } B_l$$

Partitions the support space of random vector \mathbf{x} into L regions
Represents \mathbf{x} with one of L discrete values

Illustration: 2D Vector Quantization



Represents vectors in this 2-D space with 3 reconstruction vectors

Bit-rate required after quantization $\sim \log_2 3$ per tuple $\sim 1/T(\log_2 3)$ per dimension

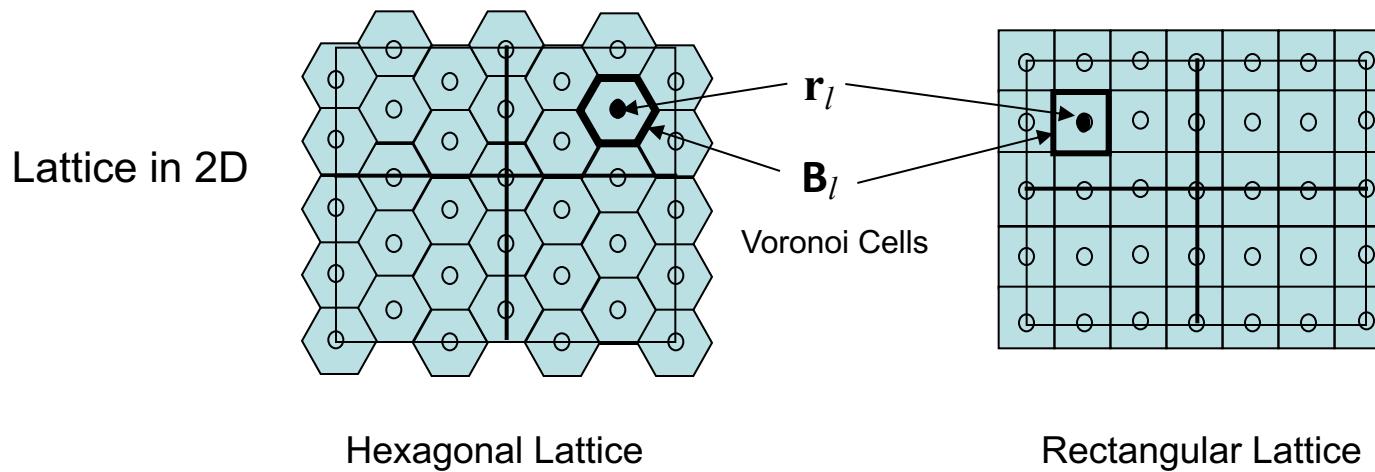
Quantization – analogous to clustering: Similar goals and algorithms

Vector Quantization Algorithms

- Linde Buzo and Gray (LBG) Algorithm
 - Analogous to the Lloyd-Max algorithm
- Choice of initial codebook
 - A representative subset of the training vectors
 - e.g., some face blocks, some shirt blocks, some background blocks
 - Scalar quantization in each dimension
 - Splitting...
- Nearest Neighbor (NN) algorithm [Equitz, 1984]
 - Start with the entire training set
 - Merge the two vectors that are closest into one vector equal to their mean
 - Repeat until the desired number of vectors is reached, or the distortion exceeds a certain threshold

Uniform Vector Quantizer

Uniform vector quantizer is a lattice quantizer



Recall: Analogy to sampling

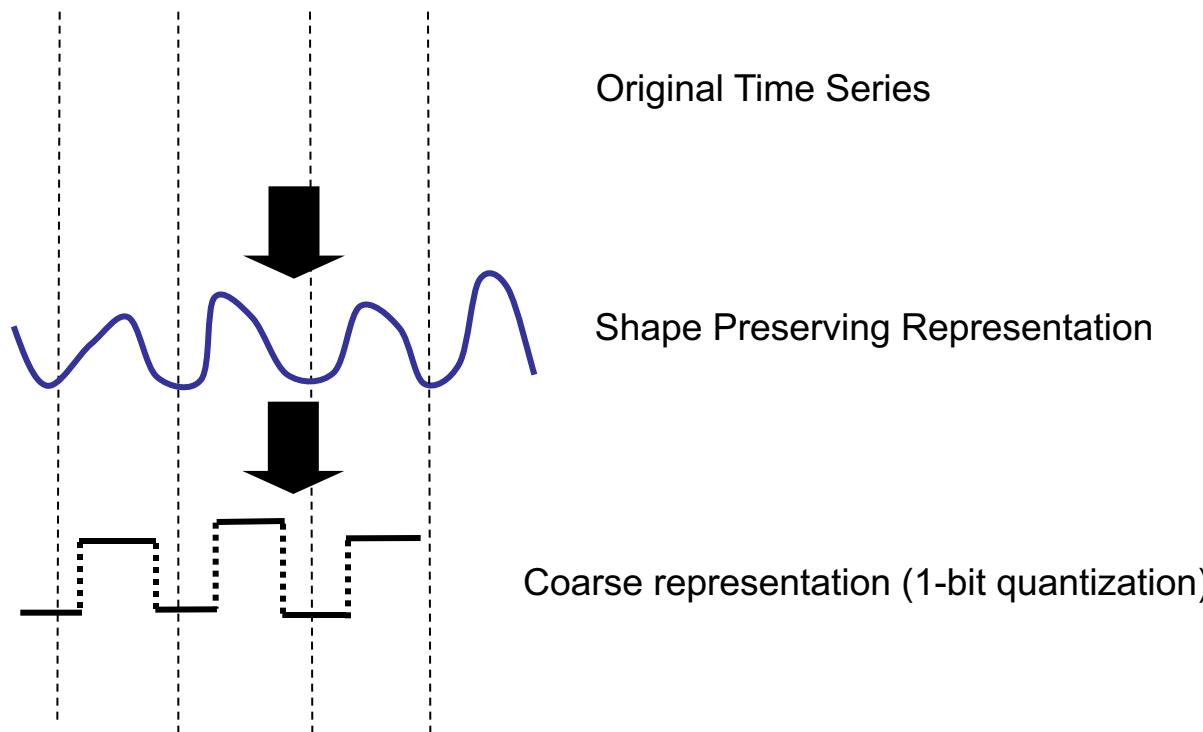
Codebook contains all lattice point (reconstruction vectors)
Nearest neighbor property used for quantization

Quantization for Stream Mining

- Given a quantizer design
 - Quantization is naturally incremental
- Quantizer design itself is not incremental
 - Can be performed within a tumbling window
- Goal may not be error in representation
 - Minimize mining specific error
 - Speaker Verification
 - ‘Shape’ preservation
 - Time series analysis and clustering

Shape Preserving Quantization

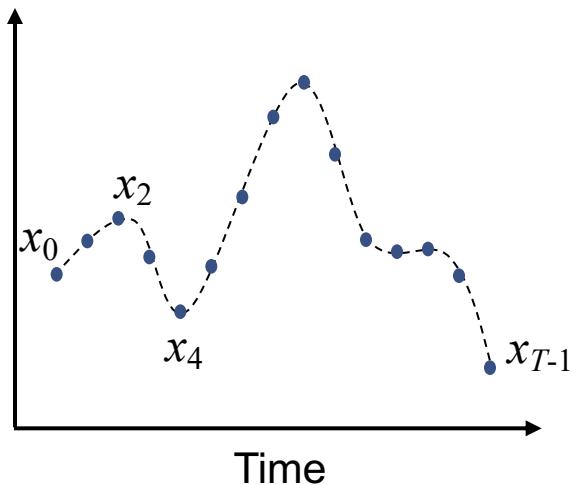
Properties of interest mostly limited to ‘shape’ of time series
Absolute values of time series samples not of interest



All three representations capture the signal periodicity faithfully

Moment Preserving Quantization

Moments – capture the shape of the signal, e.g. Mean, Variance, Skew etc.



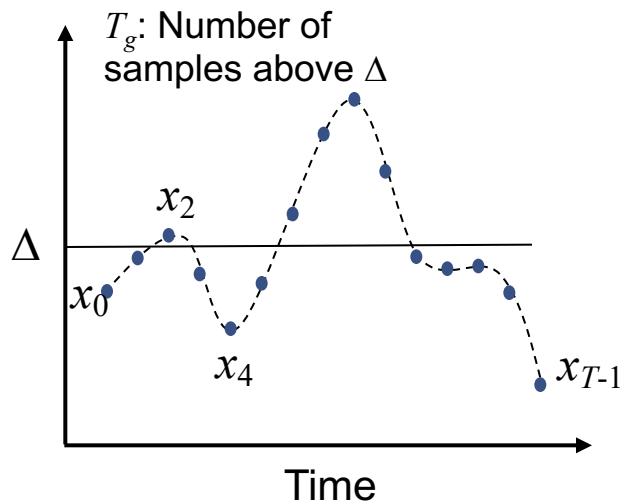
First sample moment $E[x] = \mu = \frac{1}{T} \sum_{i=0}^{T-1} x_i$

Second sample moment $E[x^2] = \sigma^2 + \mu^2 = \frac{1}{T} \sum_{i=0}^{T-1} (x_i)^2$

Consider a 1-bit quantizer \Rightarrow uses only two values to represent signal

$$Q(x_i) = \begin{cases} a & x_i \geq \Delta \\ b & x_i < \Delta \end{cases}$$

Moment Preserving Quantization



$$Q(x_i) = \begin{cases} \overbrace{\mu + \sigma \sqrt{\frac{T - T_g}{T_g}}}^a; & x_i \geq \Delta \\ \overbrace{\mu - \sigma \sqrt{\frac{T_g}{T - T_g}}}^b; & x_i < \Delta \end{cases}$$

This choice of a and b guarantees preservation of first two moments

$$\frac{1}{T} \sum_{i=0}^{T-1} Q(x_i) = \frac{1}{T} \sum_{i=0}^{T-1} x_i$$

$$\frac{1}{T} \sum_{i=0}^{T-1} [Q(x_i)]^2 = \frac{1}{T} \sum_{i=0}^{T-1} (x_i)^2$$

Moment Preserving Quantization

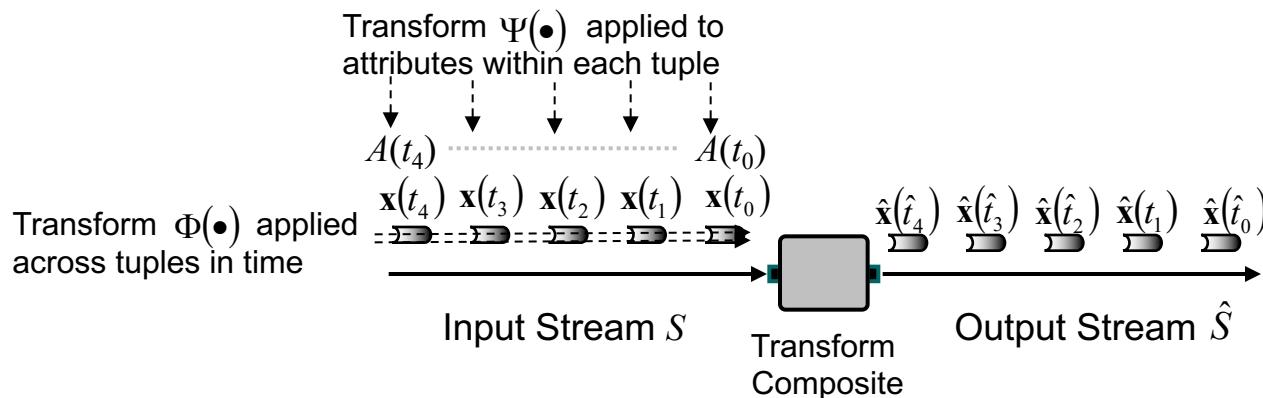
- Can select Δ to preserve third moment as well
 - Can set it to μ to avoid sending threshold
- Can be extended to B bit quantizer
 - Preserves 2^B+1 moments
- Ideal for preserving ‘stationary’ portions of signal
 - High compression rates with shape preservation
- Need to combine with change-point detection
 - To account for highly varying time series
- Windowed implementations of quantization
 - Including design of quantizer
 - Need to transmit a , b and Δ (may be implicit)
 - Different bit-allocation per time window

Quantization Summary

- Quantization used extensively in data compression
 - Rate-distortion optimization
 - Streaming: Rate-distortion-complexity optimization
- Quantizer design not incremental
 - Can be done window by window
 - *What is the impact on latency?*
- Link between vector quantization and clustering

Transforms

- Transform a tuple's attributes from one domain to another
 - Within tuple transform (for multivariate processing)
 - Across tuple transform (typically for univariate processing)
 - Temporal Transform



- Focus on Linear Transforms

1-D Transforms

$$\mathbf{x}(t_0) = \begin{bmatrix} x_0(t_0) \\ \vdots \\ x_k(t_0) \\ \vdots \\ x_{N-1}(t_0) \end{bmatrix} \quad \dots \quad \mathbf{x}(t_i) = \begin{bmatrix} x_0(t_i) \\ \vdots \\ x_k(t_i) \\ \vdots \\ x_{N-1}(t_i) \end{bmatrix} \quad \dots \quad \mathbf{x}(t_{Q-1}) = \begin{bmatrix} x_0(t_{Q-1}) \\ \vdots \\ x_k(t_{Q-1}) \\ \vdots \\ x_{N-1}(t_{Q-1}) \end{bmatrix}$$

Q numeric input tuples with N attributes each

Within Tuple Transform

$$\mathbf{x}(t_0) = \begin{bmatrix} x_0(t_0) \\ x_k(t_0) \\ x_{N-1}(t_0) \end{bmatrix} \quad \dots \quad \mathbf{x}(t_i) = \begin{bmatrix} x_0(t_i) \\ x_k(t_i) \\ x_{N-1}(t_i) \end{bmatrix} \quad \dots \quad \mathbf{x}(t_{Q-1}) = \begin{bmatrix} x_0(t_{Q-1}) \\ x_k(t_{Q-1}) \\ x_{N-1}(t_{Q-1}) \end{bmatrix}$$

$\mathbf{y} = \mathbf{x}(t_i)$
 $\Psi(\mathbf{y}) = \Lambda_{N \times N} \mathbf{x}(t_i)$

Transform operates in this direction

Temporal Transform

$$\mathbf{x}(t_0) = \begin{bmatrix} x_0(t_0) \\ \vdots \\ x_k(t_0) \\ \vdots \\ x_{N-1}(t_0) \end{bmatrix} \quad \dots \quad \mathbf{x}(t_i) = \begin{bmatrix} x_0(t_i) \\ \vdots \\ x_k(t_i) \\ \vdots \\ x_{N-1}(t_i) \end{bmatrix} \quad \dots \quad \mathbf{x}(t_{Q-1}) = \begin{bmatrix} x_0(t_{Q-1}) \\ \vdots \\ x_k(t_{Q-1}) \\ \vdots \\ x_{N-1}(t_{Q-1}) \end{bmatrix}$$

$y_i = x_k(t_i)$
 $\Phi(\mathbf{y}) = \Lambda_{Q \times Q} \mathbf{y}$

Transform operates in this direction

Refresh on 1-D Linear Transforms

What makes a transform a linear transform?

Represent signal with N samples as vector

$$\mathbf{y} = [y_0, y_1, \dots, y_{N-1}]^T$$

1-D transform represents signal as a linear combination of N basis vectors

In general, basis vector may be complex

$$\mathbf{u}_k \in C^N$$

This means $\mathbf{t} \in C^N$ even if $\mathbf{y} \in R^N$

$$\mathbf{y} = \sum_{k=0}^{N-1} t_k \mathbf{u}_k$$

basis vector (linearly independent)

transformed coefficients

With $\mathbf{t} = [t_0, t_1, \dots, t_{N-1}]^T$ and $\mathbf{U} = [\mathbf{u}_0, \mathbf{u}_1, \dots, \mathbf{u}_{N-1}]$ we have

$$\mathbf{y} = \mathbf{U}\mathbf{t}$$
 Inverse transform

and

$$\mathbf{t} = \mathbf{U}^{-1}\mathbf{y}$$
 Forward transform

Unitary 1-D Linear Transforms

Basis vectors are orthonormal

$$\langle \mathbf{u}_j, \mathbf{u}_k \rangle = \mathbf{u}_j^H \mathbf{u}_k = \begin{cases} 1 & \text{if } j = k \\ 0 & \text{otherwise} \end{cases}$$

where $\mathbf{u}_j^H = (\mathbf{u}_j^*)^T$ transpose of complex conjugate

The matrix \mathbf{U} is called unitary and we have $\mathbf{U}^H \mathbf{U} = \mathbf{U} \mathbf{U}^H = \mathbf{I}$

$\mathbf{y} = \mathbf{U}\mathbf{t}$ Inverse transform

and

$\mathbf{t} = \mathbf{U}^H \mathbf{y}$ Forward transform

1-D Fourier Transform

$$t_k = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} y_n e^{-j \frac{2\pi k n}{N}}$$

Forward transform

$$y_n = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} t_k e^{j \frac{2\pi k n}{N}}$$

Inverse transform

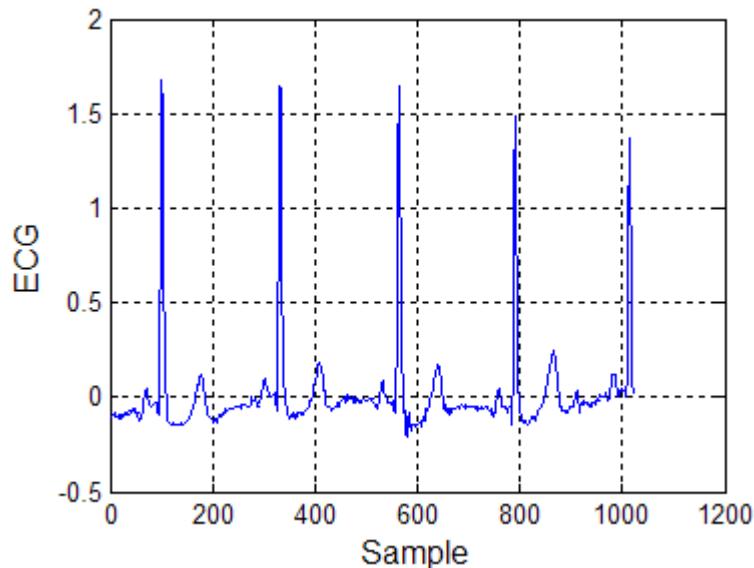
$$\mathbf{u}_k = \frac{1}{\sqrt{N}} \begin{bmatrix} 1 \\ e^{j \frac{2\pi k}{N}} \\ \vdots \\ e^{j \frac{2\pi k(N-1)}{N}} \end{bmatrix}$$

$$\mathbf{U} = \frac{1}{\sqrt{N}} \begin{bmatrix} 1 & 1 & \dots & 1 \\ 1 & e^{j \frac{2\pi}{N}} & \dots & e^{j \frac{2\pi(N-1)}{N}} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & e^{j \frac{2\pi(N-1)}{N}} & \dots & e^{j \frac{2\pi(N-1)(N-1)}{N}} \end{bmatrix}$$

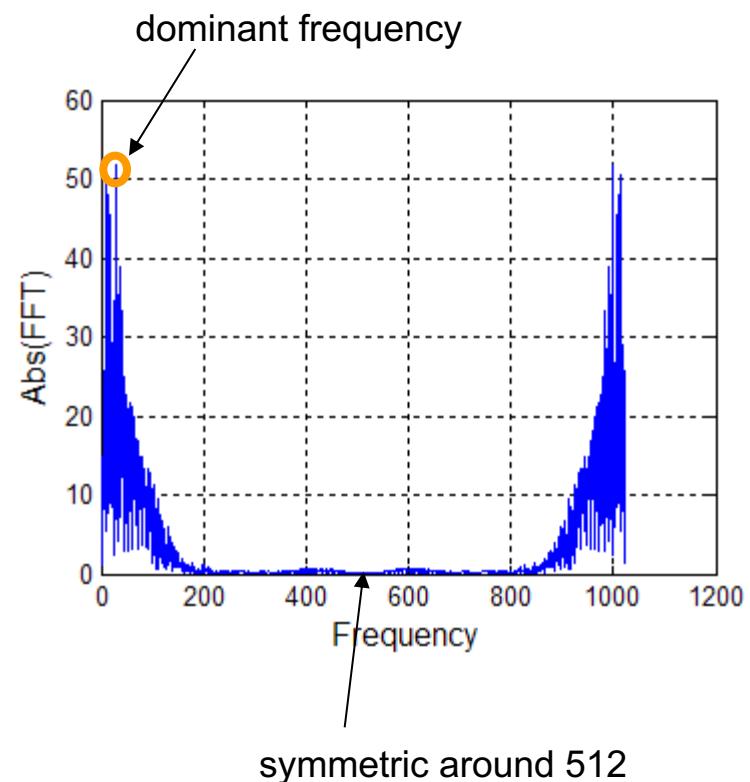
For real signal \mathbf{y} we have $t_k = t_{N-k}^*$ or symmetry in Fourier Transform

1-D Fourier Transform

For real signal \mathbf{y} , n has interpretation of time, k has interpretation of frequency



1024 point sampled ECG signal



symmetric around 512

1-D Discrete Cosine Transform

Similar* to the Fourier transform but uses only real valued bases

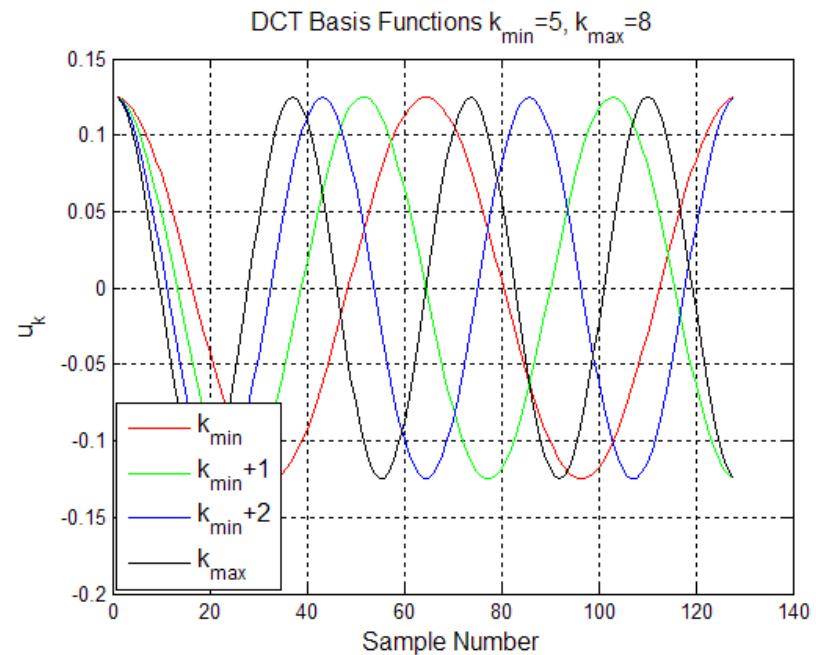
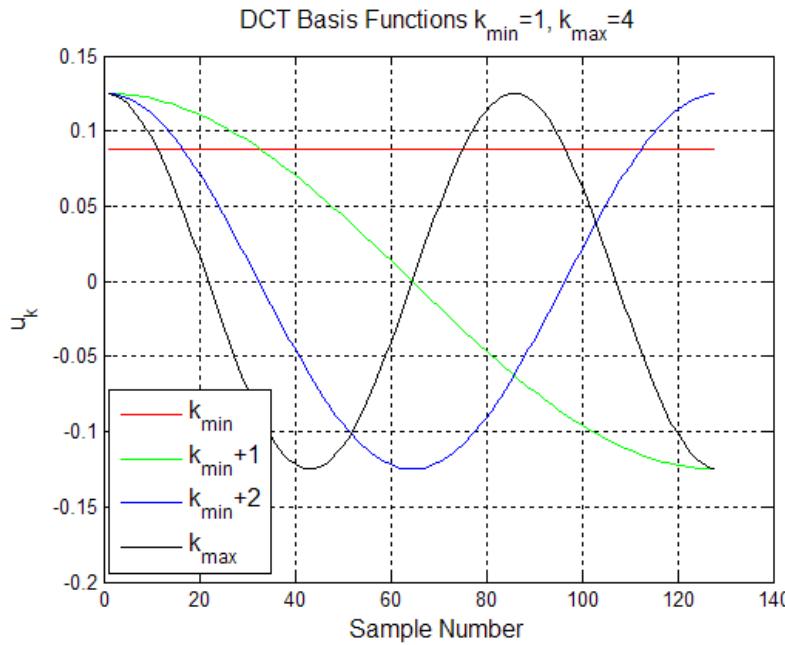
$$t_k = \sum_{n=0}^{N-1} y_n \alpha(k) \cos\left(\frac{(2n+1)k\pi}{2N}\right) \quad \text{Forward transform}$$

$$y_n = \sum_{k=0}^{N-1} t_k \alpha(k) \cos\left(\frac{(2n+1)k\pi}{2N}\right) \quad \text{Inverse transform}$$

$$\alpha(k) = \begin{cases} \frac{1}{\sqrt{N}}; & k = 0 \\ \sqrt{\frac{2}{N}}; & \text{otherwise} \end{cases}$$

$$\mathbf{U} = \frac{1}{\sqrt{N}} \begin{bmatrix} 1 & \sqrt{2} \cos\left(\frac{\pi}{2N}\right) & \dots & \sqrt{2} \cos\left(\frac{(N-1)\pi}{2N}\right) \\ 1 & \sqrt{2} \cos\left(\frac{3\pi}{2N}\right) & \dots & \sqrt{2} \cos\left(\frac{3(N-1)\pi}{2N}\right) \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \sqrt{2} \cos\left(\frac{(2N-1)\pi}{2N}\right) & \dots & \sqrt{2} \cos\left(\frac{(2N-1)(N-1)\pi}{2N}\right) \end{bmatrix}$$

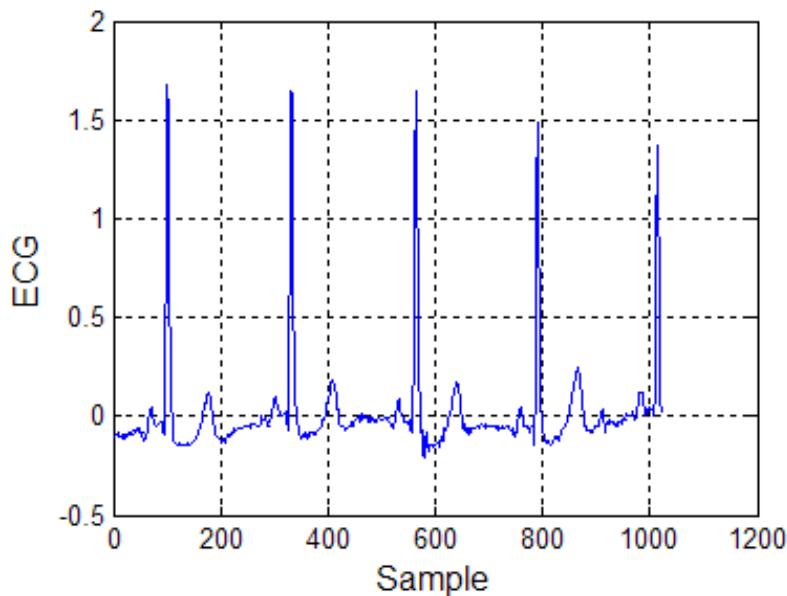
1-D Discrete Cosine Transform



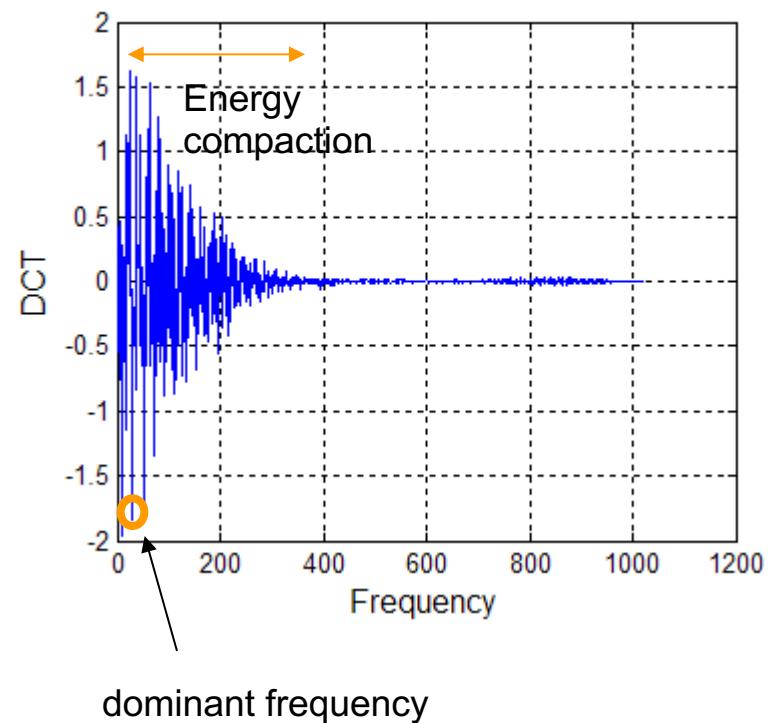
First 8 basis vectors for $N=128$

Basis functions are sinusoids with increasing frequency

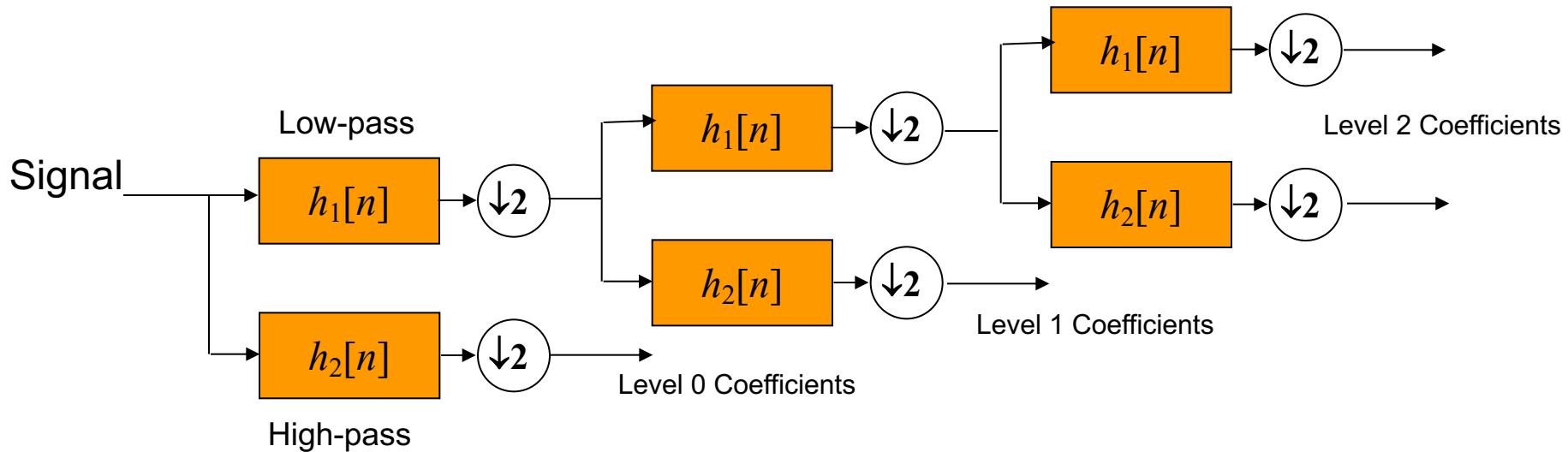
1-D Discrete Cosine Transform



1024 point sampled ECG signal

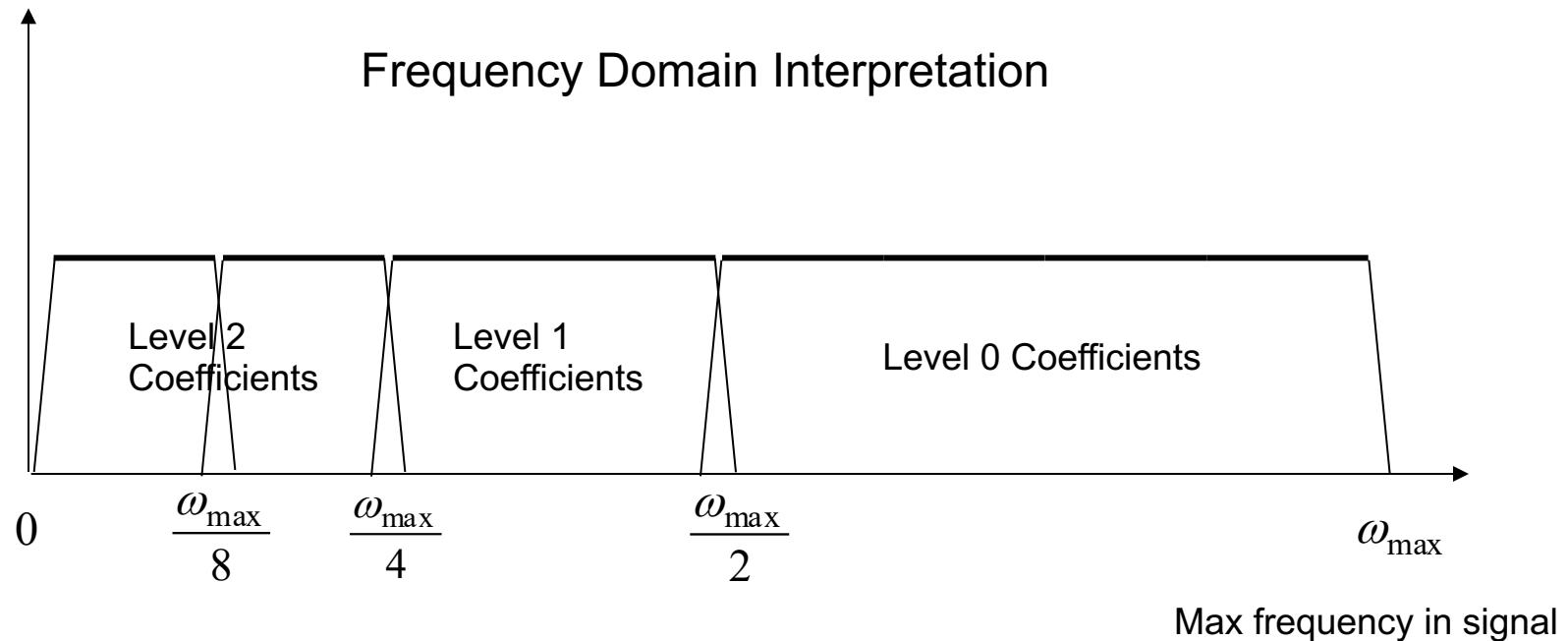


Subband Coding: Wavelet Transform



Recursively apply decomposition to low freq band \Rightarrow increase frequency resolution
Set of filters collectively called – “Filter bank”
Multi-resolution representation of signal – **Scalable Coding**

Subband Coding: Wavelet Transform



Wavelet Transform: Recursively partitions frequency space
Provides time-frequency localization

1-D Haar Wavelet Transform

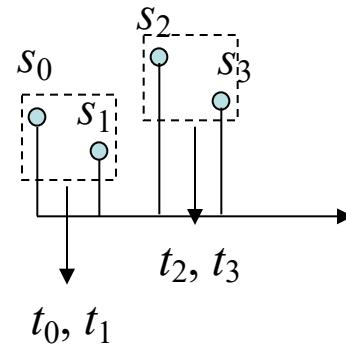
$$\mathbf{U} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

$$\mathbf{U}^H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

$$t_0 = \frac{s_0 + s_1}{\sqrt{2}}$$

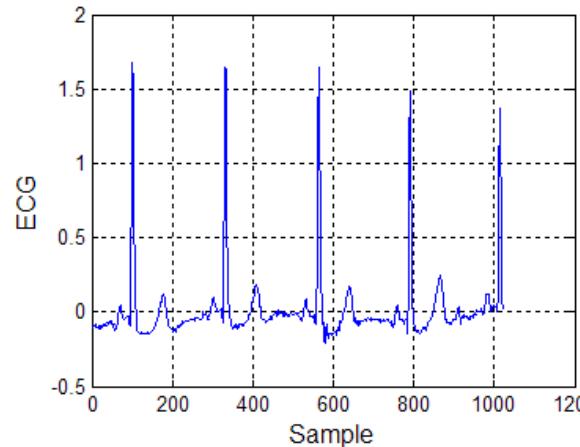
$$t_1 = \frac{s_0 - s_1}{\sqrt{2}}$$

Normalized sum
and difference

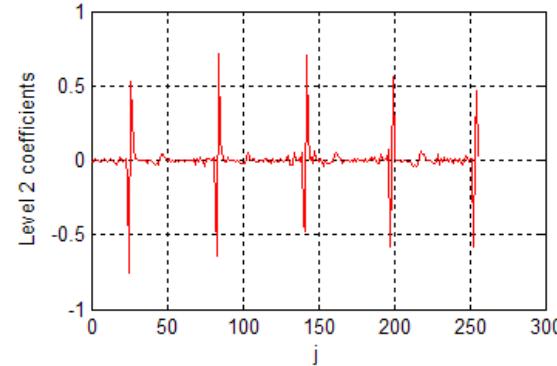
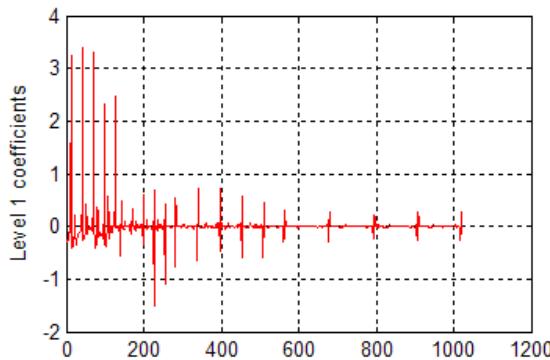


Widnowing Based
Implementation

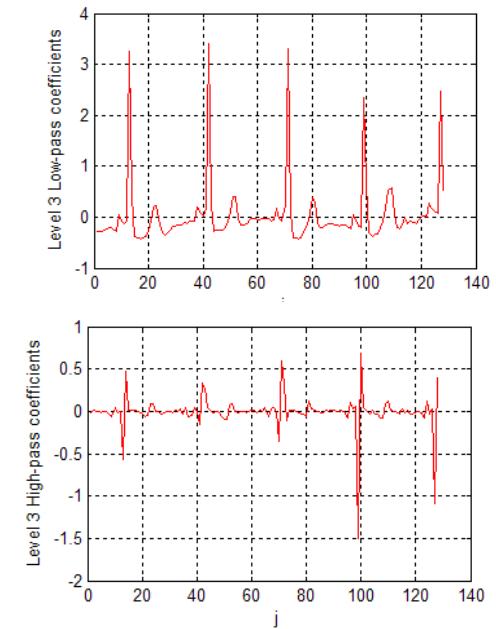
1-D Haar Wavelet Transform



Original
Signal



Level 2 Coefficients

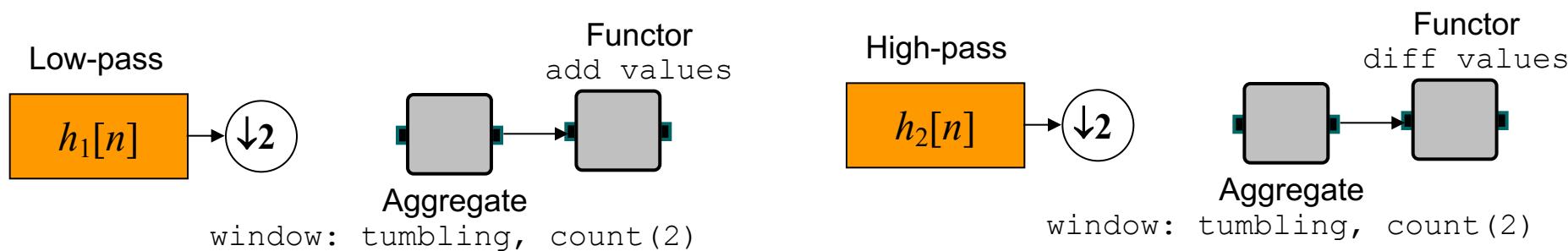
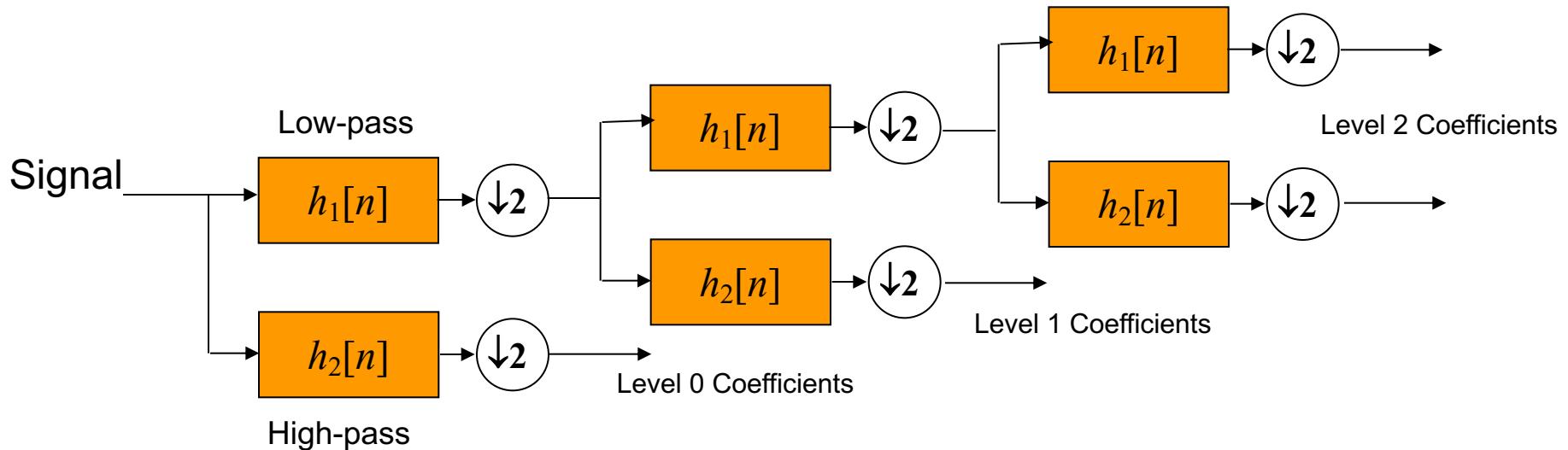


Level 3 Coefficients

Haar Wavelet Transform
How does this relate to compressibility?

Implementing the Temporal Haar Transform

Applied window by window: For a given window, with a fixed size



Implementing the Temporal Haar Transform

- Fixed structure
 - if window size fixed, and known apriori can construct topology at compile time
- What can we do if window size unknown apriori?
 - Construct a transform matrix on the fly
 - Consider a 2 level Haar Transform. *Can we build a single transform matrix for this?*

$$U = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

$$\Phi = \frac{1}{2} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ \sqrt{2} & \sqrt{2} & 0 & 0 \\ 0 & 0 & \sqrt{2} & -\sqrt{2} \end{bmatrix}$$

Implementing the Haar Transform

- Temporal transform with unknown window size
 - Aggregate tuples for the window
 - Construct the transform matrix
 - Apply across the tuples (or tuple attributes)
- Within tuple transform
 - Can be applied tuple by tuple
 - Purely streaming – with transform matrix approach
 - Do not need any aggregation

From 1-D to 2-D Transforms

2-D Transform: Across tuples and across attributes within tuple

$$\Lambda_{N \times N} \begin{bmatrix} x_0(t_0) & x_0(t_i) & x_0(t_{Q-1}) \\ \vdots & \vdots & \vdots \\ x_k(t_0) & x_k(t_i) & x_k(t_{Q-1}) \\ \vdots & \vdots & \vdots \\ x_{N-1}(t_0) & x_{N-1}(t_i) & x_{N-1}(t_{Q-1}) \\ \vdots & & \end{bmatrix} \Lambda_{Q \times Q}^T$$

Within tuple transform Temporal transform

Collect tuples into one matrix $\mathbf{X}_{N \times Q}$

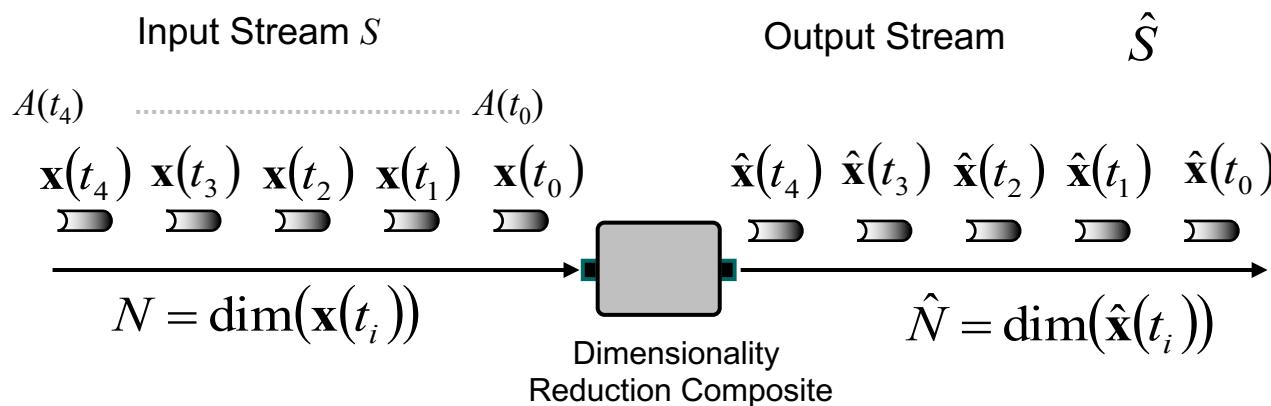
Assume Separability

Transforms: Summary

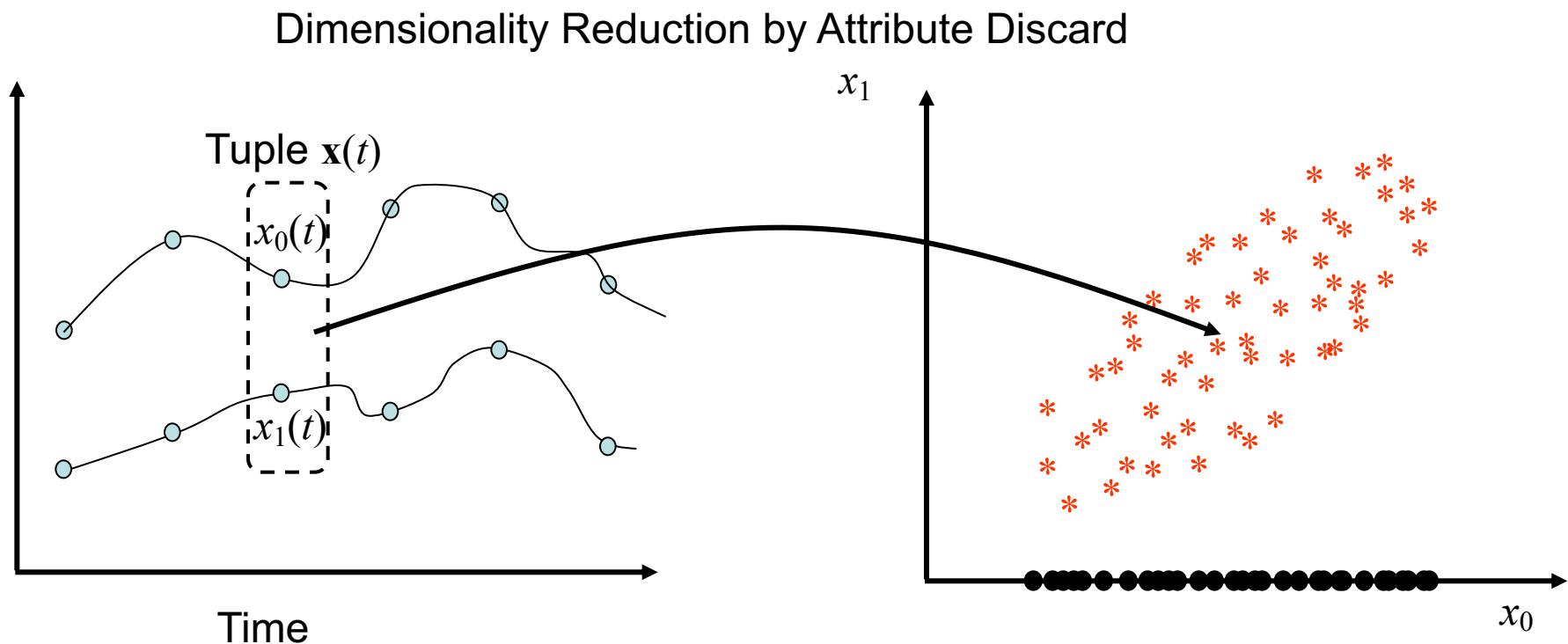
- Capture time-frequency properties of the signal
 - Allow different features to be extracted
- Allow approximating signal with smaller number of coefficients
 - Energy compaction
 - Link to dimensionality reduction and sketches
- Linear transforms
 - Provide windowed computation
 - Provide reversible representations

Dimensionality Reduction

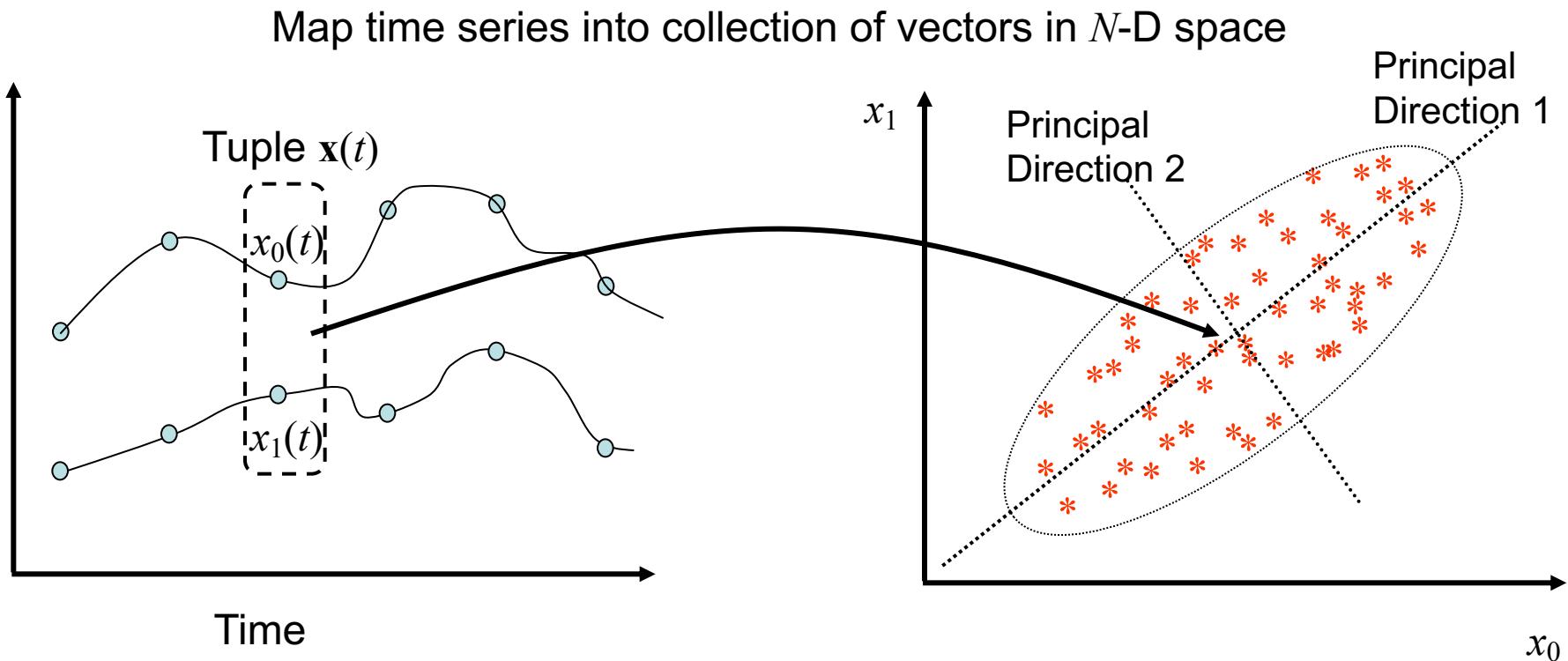
- Applied within tuple
- Reduce number of attributes within tuple
 - From N to \hat{N}
- For numeric tuple
 - Appropriately transform the data into domain where attributes can be discarded
 - See link with energy compacting transforms



Dimensionality Reduction 2-D Example



Dimensionality Reduction: 2-D Example



PCA identifies principal directions of variation in signal

Refresh: PCA and Karhunen-Loewe Transform

- Special kind of unitary transforms
 - Basis vectors are eigenvectors of covariance matrix
 - Decorrelates the dimensions
- Provides guaranteed smallest average squared reconstruction error
 - When only $K (< N)$ coefficients are used to approximate signal
 - Consider that we look at Q time steps

$$\boldsymbol{\mu}_S = \frac{1}{Q} \sum_{j=0}^{Q-1} \mathbf{x}(t_j)$$

Sample Mean vector

$$\boldsymbol{\Sigma}_S = \frac{1}{Q} \sum_{j=0}^{Q-1} (\mathbf{x}(t_j) - \boldsymbol{\mu})(\mathbf{x}(t_j) - \boldsymbol{\mu})^T$$

Sample Covariance matrix

PCA and KLT

The basis vectors for KLT are eigenvectors of the covariance matrix of \mathbf{S}

$$\Sigma_S \mathbf{u}_k = \lambda_k \mathbf{u}_k$$

Using the relationship $\Sigma_T = \mathbf{U}^H \Sigma_S \mathbf{U}$ we have

$$\Sigma_T = \begin{bmatrix} \mathbf{u}_0^H \\ \vdots \\ \mathbf{u}_{N-1}^H \end{bmatrix} \Sigma_S [\mathbf{u}_0 \quad \dots \quad \mathbf{u}_{N-1}]$$
$$\Sigma = \begin{bmatrix} \mathbf{u}_0^H \\ \vdots \\ \mathbf{u}_{N-1}^H \end{bmatrix} [\lambda_0 \mathbf{u}_0 \quad \dots \quad \lambda_{N-1} \mathbf{u}_{N-1}] = \begin{bmatrix} \lambda_0 & 0 & \dots & 0 \\ 0 & \lambda_1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \lambda_{N-1} \end{bmatrix}$$

Diagonal Matrix
Decorrelating transform

The eigenvalues are equal to the variance of transformed coefficients $\lambda_k = \sigma_{t_k}^2$

PCA and KLT

- Provides optimal squared error representation
 - Very efficient dimensionality reduction
 - Added de-correlating property
- Hard to compute in practice
 - Need all vectors available
 - Need to compute covariance matrix
 - Need to compute eigenvectors
 - Not directly suited for streaming data
- Incremental PCA algorithms
 - Approximate PCA

SPIRIT Algorithm

- SPIRIT: Streaming Pattern discoveRy in multiple Timeseries
- Transform based dimensionality reduction
 - Online algorithm: Compute the basis vectors (eigenvectors) incrementally instead of doing an eigen-decomposition
 - Can change amount of reduction incrementally
- Used in multiple applications
 - Forecasting
 - Pattern Detection
 - Anomaly Detection

SPIRIT Algorithm

Input dimensions N
Output dimensions \hat{N}

Basis vectors
 $\mathbf{u}_k = \begin{bmatrix} \vdots \\ 1 \\ \vdots \end{bmatrix}$

Energy (eigenvalue)
 $d_k = \varepsilon$
 $0 \leq k \leq \hat{N}-1$

Initialization
 $\mathbf{z} = \mathbf{x}(t)$

for $k=0: \hat{N}-1$

New Tuple

Iterate through the reduced number of dimensions

$\lambda_k = (\mathbf{z})^T \mathbf{u}_k$

Project onto k -th basis vector

**Iterative
Update**

$d_k = (\lambda_k)^2 + \alpha d_k$

Update energy of k -th basis vector

SPIRIT Algorithm

Iterative Update

$$\mathbf{z} = \mathbf{x}(t)$$

for $k=0:\hat{N}-1$

$$\lambda_k = (\mathbf{z})^T \mathbf{u}_k$$

$$d_k = (\lambda_k)^2 + \alpha d_k$$

$$\mathbf{e}_k = \mathbf{z} - \lambda_k \mathbf{u}_k$$

$$\mathbf{u}_k = \mathbf{u}_k + \frac{1}{d_k} \lambda_k \mathbf{e}_k$$

$$\mathbf{z} = \mathbf{z} - \lambda_k \mathbf{u}_k$$

end

submit vector $[\lambda_0 \dots \lambda_{\hat{N}-1}]$

New Tuple

Iterate through the reduced number of dimensions

Project onto k -th basis vector

Update energy of k -th basis vector

Find approximation error

Update basis vector

Update basis vector

Dimensionality Reduction

- SPIRIT
 - Can produce approximations to PCA
 - Number of vectors can be changed incrementally (based on error measures)
- Other techniques
 - MUSCLES
 - Distributed Dimensionality Reduction
 - SVD based dimensionality reduction

Summary

Techniques	Tuple Types	Windowing
Sketches	Numeric or non-numeric Tuples, Univariate or Multivariate	Not windowed. Can be tumbling window
Quantization	Numeric tuples. Univariate (scalar) or Multivariate (vector)	Tumbling window based
Transforms	Numeric tuples. Univariate (across tuples) or multivariate (within tuples) or both	Not windowed. Temporal transforms typically tumbling window
Dimensionality Reduction	Numeric tuples. Multivariate	Not windowed. Can be tumbling window

Wrapup

- Pre-processing Algorithms
 - Sketches
 - Quantization
 - Transforms
 - Dimensionality Reduction
- Next lecture
 - Stream mining and modeling algorithms
 - Classification
 - Regression

Reading

- Book – Chapter 10: Preprocessing Algorithms
- Other Reading
 - Sampling and Summarization: Data Streams: Models and Algorithms (Chapter 2, Chapter 9, Chapter 12)
 - Sampling (Uniform, Non-uniform, threshold based)
 - Signal Processing Text
 - Reservoir Sampling
 - <http://www.cs.umd.edu/~samir/498/vitter.pdf>
 - Linear Transforms
 - Signal Processing Text (Alan V. Oppenheim, Ronald W. Schafer, John R. Buck : Discrete-Time Signal Processing)
 - Quantization: Signal Processing Text
 - Task specific Quantization for Speaker Verification
 - H. Tseng et al, “Quantization for Adapted GMM-based speaker verification”, ICASSP 2006.
 - Moment Preserving Quantization
 - E. Delp, M. Saenz, and P. Salama. "Block Truncation Coding (BTC)," The Handbook of Image and Video Processing. Academic Press, 2000.

Streaming Algorithms II

Preprocessing and Transformation

Objectives

- Feedback on Seminars
- Projects
- Brief Recap
- Streaming Algorithms for Pre-processing and Transformation
 - Transforms
 - Dimensionality Reduction
- Intro to Modeling

Aside: Projects

- <https://sites.google.com/site/fundamentalsofstreamprocessing/projects>
- Data/Input
 - What data sources will you use?
 - Will the data be stored and replayed, or pulled in live?
 - Do you need to create new connectors to access the data?
- Techniques/Application
 - What is the problem you will solve?
 - Is there a set of references that motivate this?
 - Do you need to integrate other tools?
- Results
 - How will you present your results?
 - What will you show as a demo?
 - What techniques will you plan to use?
- Next Steps
 - Will you use any algorithms from class? Will you use any optimizations

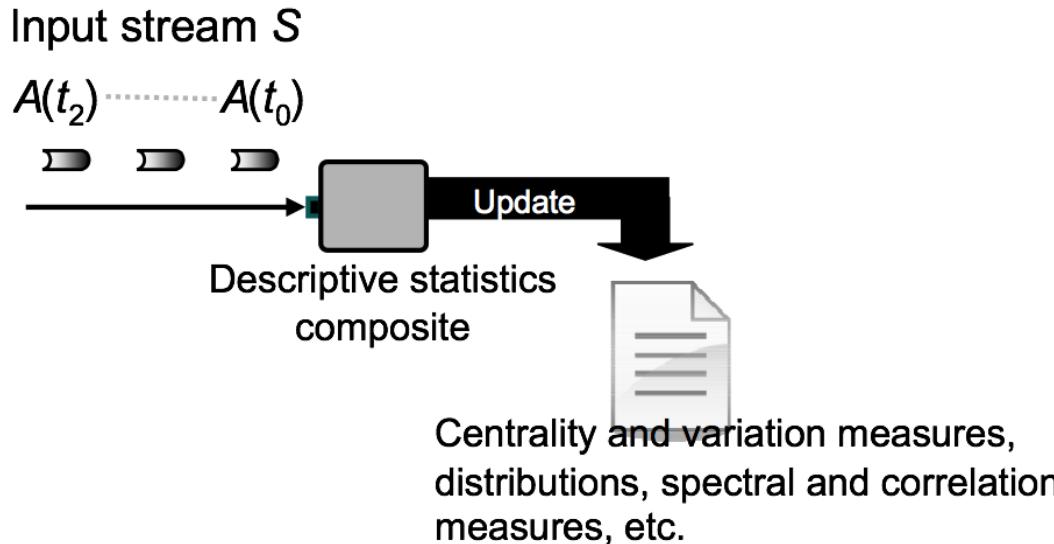
Recap: Outline of Data Mining Process

- Data acquisition
 - Collect data from external sources
- Data pre-processing
 - Prepare data for further analysis: data cleaning, data interpolation (missing data), data normalization (heterogeneous sources), temporal alignment and data formatting, data reduction
- Data transformation
 - Select an appropriate representation for the data
 - Select or compute the relevant features (*feature extraction/selection*)
- Modeling (data mining)
 - Identify interesting patterns, similarity and groupings, partition into classes, fit functions, find dependencies and correlations, identifying abnormal data
- Evaluation
 - Use of the mining model and evaluation of the results

Recap: Data Preprocessing and Transformation

- Descriptive statistics
 - Extracting simple quantitative statistics of the distribution
- Sampling
 - Reducing the volume of the input data by retaining only an appropriate subset for analysis
- Sketches
 - Compact data structures that contain synopses of the streaming data, for approximate query answering
- Quantization
 - Reduce the fidelity of individual data samples to lower compute and memory costs
- Dimensionality reduction
 - Reduce the number of attributes within each tuple to decrease data volume and improve accuracy of the models
- Transforms
 - Convert data items or tuples and their attributes from one domain to another, such that data is better suited for further analysis

Descriptive Statistics



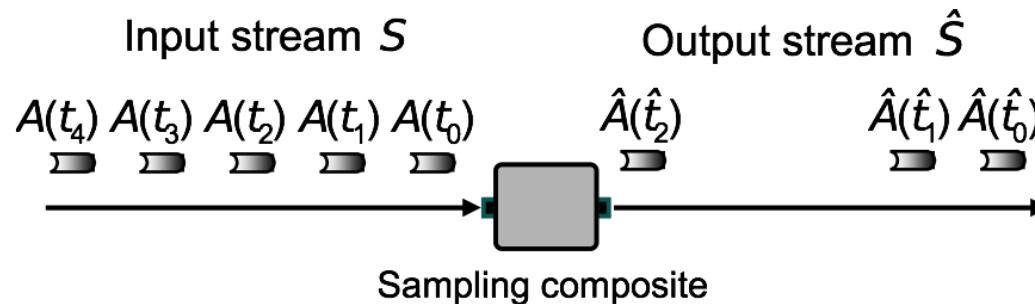
- Extract information from the tuples in a stream such that
 - statistical properties of the stream can be characterized or summarized
 - the quality of the data it transports assessed

Descriptive Stats: BasicCounting

- Question: Assume you have a stream that contains tuples that are either 0s or 1s, how can you maintain the number of 1s in the last W tuples
 - Sliding window of size W , slide 1
- Assume W is large
 - So $O(W)$ is too large. We want to store logarithmic space
- The *BasicCounting* algorithm
 - Uses space of the order $O\left(\frac{1}{\varepsilon} \log^2 W\right)$
 - With error tolerance ε , i.e. the counts are within $1 \pm \varepsilon$ of its actual value. ε is specified by application objective
- Study: Data structure, update, query

Sampling

- Sampling is used to reduce the data stream by selecting some tuples based on one or more criteria



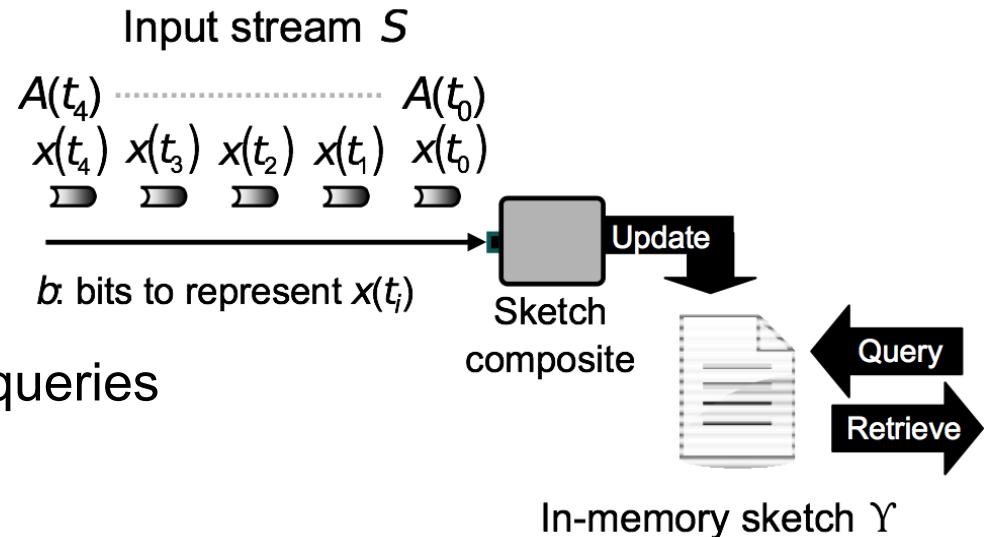
- Three types of sampling
 - Systematic or Uniform
 - Data Driven
 - Random

Reservoir Sampling Algorithm

- Let q be the reservoir size
 - Indices in reservoir run from 0 to $q-1$
- Let i be the index of the tuple being processed right now
- If $i < q$
 - Append the tuple to the reservoir
- Otherwise
 - Select p as a random integer in range $[0, i]$
 - If $0 \leq p < q$
 - Replace tuple at index p in reservoir with current tuple

Sketches

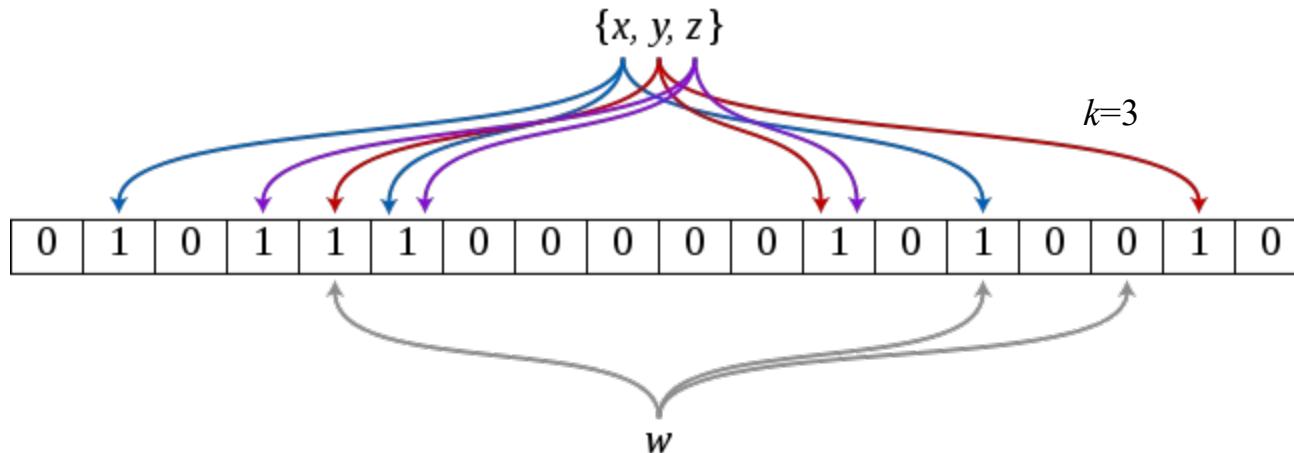
- In-memory data structures that contain compact, often lossy, synopses of streaming data
- They capture the key properties of the stream



- In order to answer specific queries
 - Error bounds and
 - probabilistic guarantees
- Used for approximate query processing

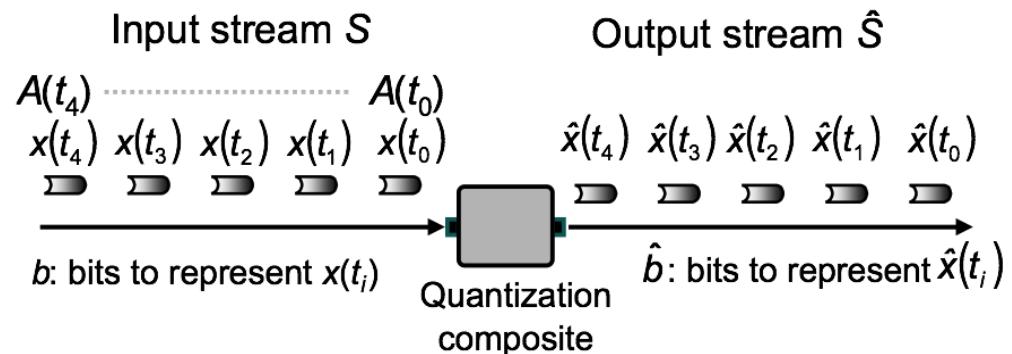
Bloom Filters

- Let us say we want to answer containment queries
- Whether a given item has been seen so far or not
- Bloom filter
 - Keep a bit array of size m
 - Use k pairwise independent hash functions (more later)
 - Update: Hash item to k locations and set the bits to one
 - Query: Hash item to k locations
 - Return true if all of the k locations are set

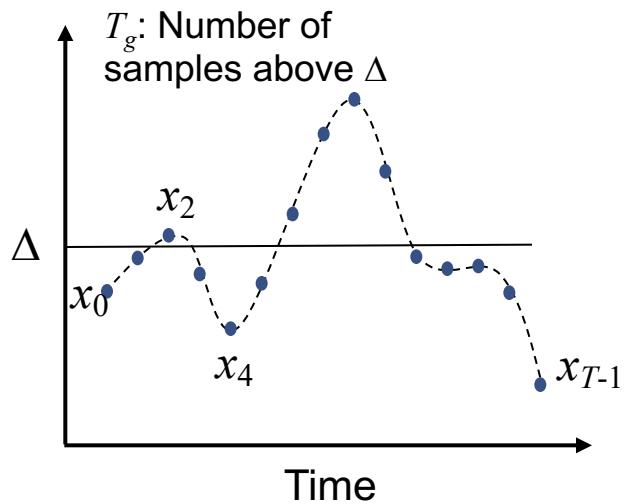


Quantization

- Simplest lossy data reduction mechanism
 - Perfect reconstruction of original signal not possible
- Maps Input Sample → Reconstruction Value from predefined codebook
- Example
 - Rounding Function
 - Floor Function
 - Truncation
- Original sample
 - Numeric Tuples with Continuous or Discrete Space



Moment Preserving Quantization



$$Q(x_i) = \begin{cases} \overbrace{\mu + \sigma \sqrt{\frac{T - T_g}{T_g}}}^a; & x_i \geq \Delta \\ \overbrace{\mu - \sigma \sqrt{\frac{T_g}{T - T_g}}}^b; & x_i < \Delta \end{cases}$$

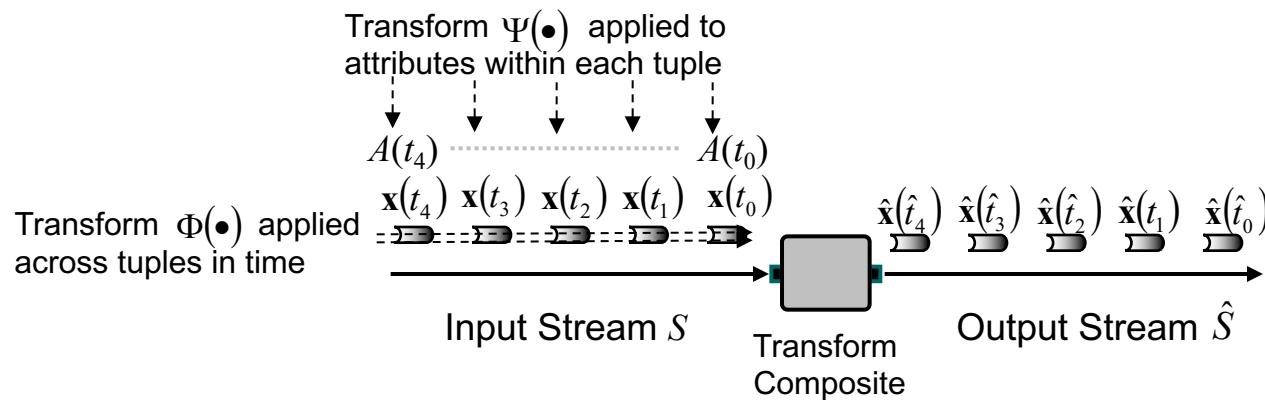
This choice of a and b guarantees preservation of first two moments

$$\frac{1}{T} \sum_{i=0}^{T-1} Q(x_i) = \frac{1}{T} \sum_{i=0}^{T-1} x_i$$

$$\frac{1}{T} \sum_{i=0}^{T-1} [Q(x_i)]^2 = \frac{1}{T} \sum_{i=0}^{T-1} (x_i)^2$$

Transforms

- Transform a tuple's attributes from one domain to another
 - Within tuple transform (for multivariate processing)
 - Across tuple transform (typically for univariate processing)
 - Temporal Transform



- Focus on Linear Transforms

1-D Transforms

$$\mathbf{x}(t_0) = \begin{bmatrix} x_0(t_0) \\ \vdots \\ x_k(t_0) \\ \vdots \\ x_{N-1}(t_0) \end{bmatrix} \quad \dots \quad \mathbf{x}(t_i) = \begin{bmatrix} x_0(t_i) \\ \vdots \\ x_k(t_i) \\ \vdots \\ x_{N-1}(t_i) \end{bmatrix} \quad \dots \quad \mathbf{x}(t_{Q-1}) = \begin{bmatrix} x_0(t_{Q-1}) \\ \vdots \\ x_k(t_{Q-1}) \\ \vdots \\ x_{N-1}(t_{Q-1}) \end{bmatrix}$$

Q numeric input tuples with N attributes each

Within Tuple Transform

$$\mathbf{x}(t_0) = \begin{bmatrix} x_0(t_0) \\ x_k(t_0) \\ x_{N-1}(t_0) \end{bmatrix} \quad \dots \quad \mathbf{x}(t_i) = \begin{bmatrix} x_0(t_i) \\ x_k(t_i) \\ x_{N-1}(t_i) \end{bmatrix} \quad \dots \quad \mathbf{x}(t_{Q-1}) = \begin{bmatrix} x_0(t_{Q-1}) \\ x_k(t_{Q-1}) \\ x_{N-1}(t_{Q-1}) \end{bmatrix}$$

$\mathbf{y} = \mathbf{x}(t_i)$
 $\Psi(\mathbf{y}) = \Lambda_{N \times N} \mathbf{x}(t_i)$

Transform operates in this direction

Temporal Transform

$$\mathbf{x}(t_0) = \begin{bmatrix} x_0(t_0) \\ \vdots \\ x_k(t_0) \\ \vdots \\ x_{N-1}(t_0) \end{bmatrix} \quad \dots \quad \mathbf{x}(t_i) = \begin{bmatrix} x_0(t_i) \\ \vdots \\ x_k(t_i) \\ \vdots \\ x_{N-1}(t_i) \end{bmatrix} \quad \dots \quad \mathbf{x}(t_{Q-1}) = \begin{bmatrix} x_0(t_{Q-1}) \\ \vdots \\ x_k(t_{Q-1}) \\ \vdots \\ x_{N-1}(t_{Q-1}) \end{bmatrix}$$

$y_i = x_k(t_i)$
 $\Phi(\mathbf{y}) = \Lambda_{Q \times Q} \mathbf{y}$

Transform operates in this direction

Refresh on 1-D Linear Transforms

What makes a transform a linear transform?

Represent signal with N samples as vector

$$\mathbf{y} = [y_0, y_1, \dots, y_{N-1}]^T$$

1-D transform represents signal as a linear combination of N basis vectors

In general, basis vector may be complex

$$\mathbf{u}_k \in C^N$$

This means $\mathbf{t} \in C^N$ even if $\mathbf{y} \in R^N$

$$\mathbf{y} = \sum_{k=0}^{N-1} t_k \mathbf{u}_k$$

basis vector (linearly independent)

transformed coefficients

With $\mathbf{t} = [t_0, t_1, \dots, t_{N-1}]^T$ and $\mathbf{U} = [\mathbf{u}_0, \mathbf{u}_1, \dots, \mathbf{u}_{N-1}]$ we have

$$\mathbf{y} = \mathbf{U}\mathbf{t}$$
 Inverse transform

and

$$\mathbf{t} = \mathbf{U}^{-1}\mathbf{y}$$
 Forward transform

Unitary 1-D Linear Transforms

Basis vectors are orthonormal

$$\langle \mathbf{u}_j, \mathbf{u}_k \rangle = \mathbf{u}_j^H \mathbf{u}_k = \begin{cases} 1 & \text{if } j = k \\ 0 & \text{otherwise} \end{cases}$$

where $\mathbf{u}_j^H = (\mathbf{u}_j^*)^T$ transpose of complex conjugate

The matrix \mathbf{U} is called unitary and we have $\mathbf{U}^H \mathbf{U} = \mathbf{U} \mathbf{U}^H = \mathbf{I}$

$\mathbf{y} = \mathbf{U}\mathbf{t}$ Inverse transform

and

$\mathbf{t} = \mathbf{U}^H \mathbf{y}$ Forward transform

1-D Fourier Transform

$$t_k = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} y_n e^{-j \frac{2\pi k n}{N}}$$

Forward transform

$$y_n = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} t_k e^{j \frac{2\pi k n}{N}}$$

Inverse transform

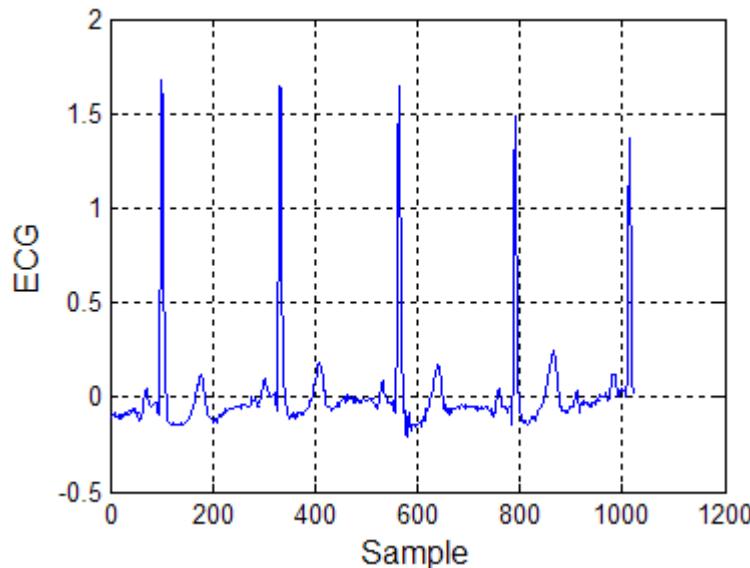
$$\mathbf{u}_k = \frac{1}{\sqrt{N}} \begin{bmatrix} 1 \\ e^{j \frac{2\pi k}{N}} \\ \vdots \\ e^{j \frac{2\pi k(N-1)}{N}} \end{bmatrix}$$

$$\mathbf{U} = \frac{1}{\sqrt{N}} \begin{bmatrix} 1 & 1 & \dots & 1 \\ 1 & e^{j \frac{2\pi}{N}} & \dots & e^{j \frac{2\pi(N-1)}{N}} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & e^{j \frac{2\pi(N-1)}{N}} & \dots & e^{j \frac{2\pi(N-1)(N-1)}{N}} \end{bmatrix}$$

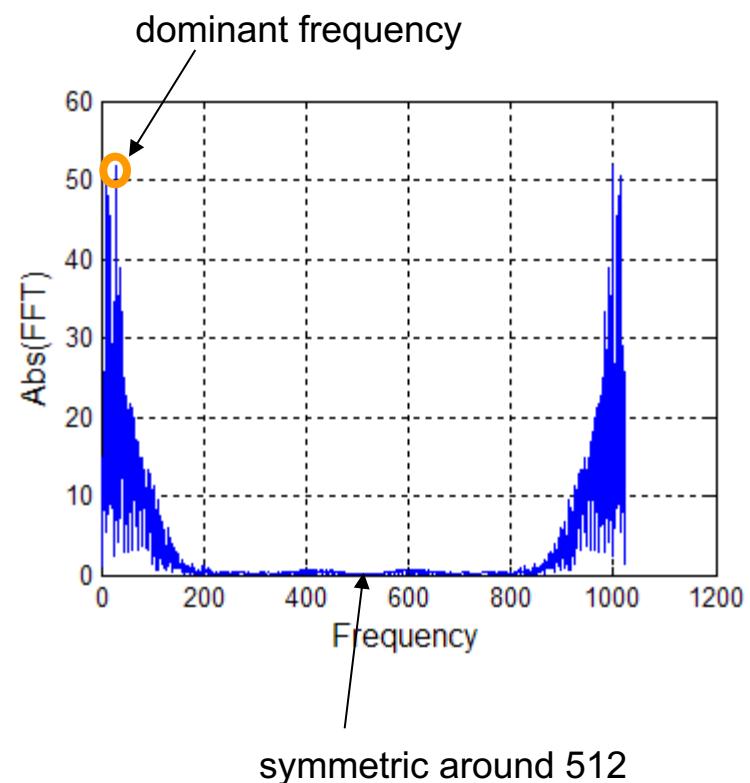
For real signal \mathbf{y} we have $t_k^* = t_{N-k}$ or symmetry in Fourier Transform

1-D Fourier Transform

For real signal \mathbf{y} , n has interpretation of time, k has interpretation of frequency



1024 point sampled ECG signal



1-D Discrete Cosine Transform

Similar* to the Fourier transform but uses only real valued bases

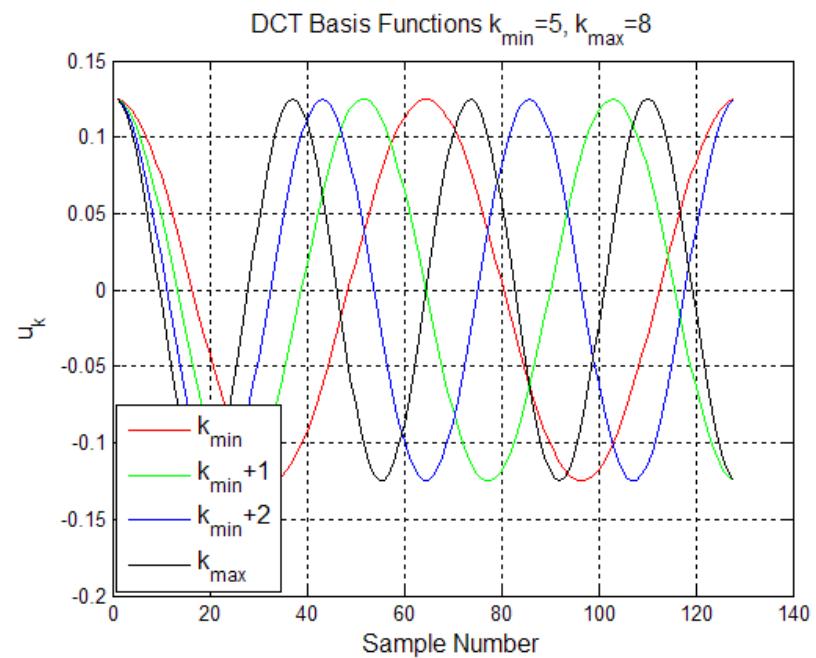
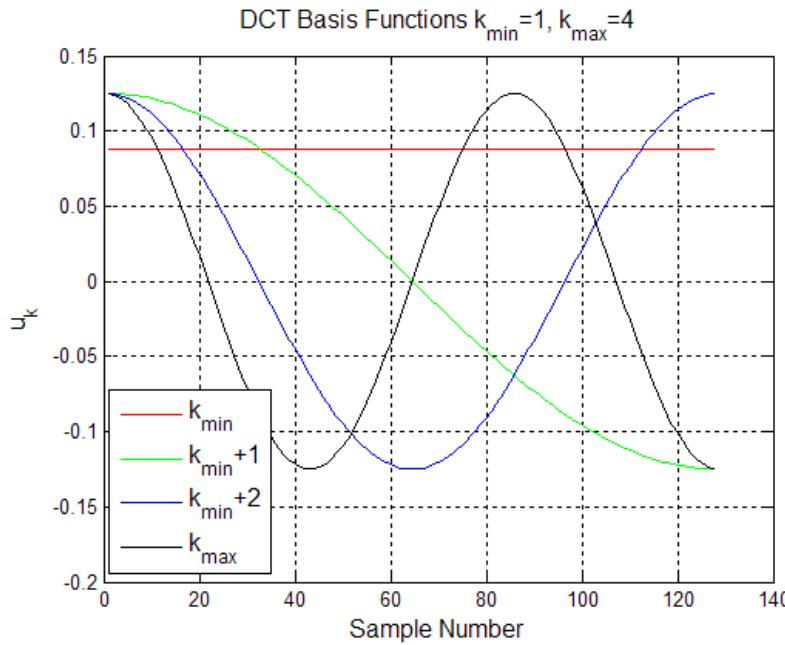
$$t_k = \sum_{n=0}^{N-1} y_n \alpha(k) \cos\left(\frac{(2n+1)k\pi}{2N}\right) \quad \text{Forward transform}$$

$$y_n = \sum_{k=0}^{N-1} t_k \alpha(k) \cos\left(\frac{(2n+1)k\pi}{2N}\right) \quad \text{Inverse transform}$$

$$\alpha(k) = \begin{cases} \frac{1}{\sqrt{N}}; & k = 0 \\ \sqrt{\frac{2}{N}}; & \text{otherwise} \end{cases}$$

$$\mathbf{U} = \frac{1}{\sqrt{N}} \begin{bmatrix} 1 & \sqrt{2} \cos\left(\frac{\pi}{2N}\right) & \dots & \sqrt{2} \cos\left(\frac{(N-1)\pi}{2N}\right) \\ 1 & \sqrt{2} \cos\left(\frac{3\pi}{2N}\right) & \dots & \sqrt{2} \cos\left(\frac{3(N-1)\pi}{2N}\right) \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \sqrt{2} \cos\left(\frac{(2N-1)\pi}{2N}\right) & \dots & \sqrt{2} \cos\left(\frac{(2N-1)(N-1)\pi}{2N}\right) \end{bmatrix}$$

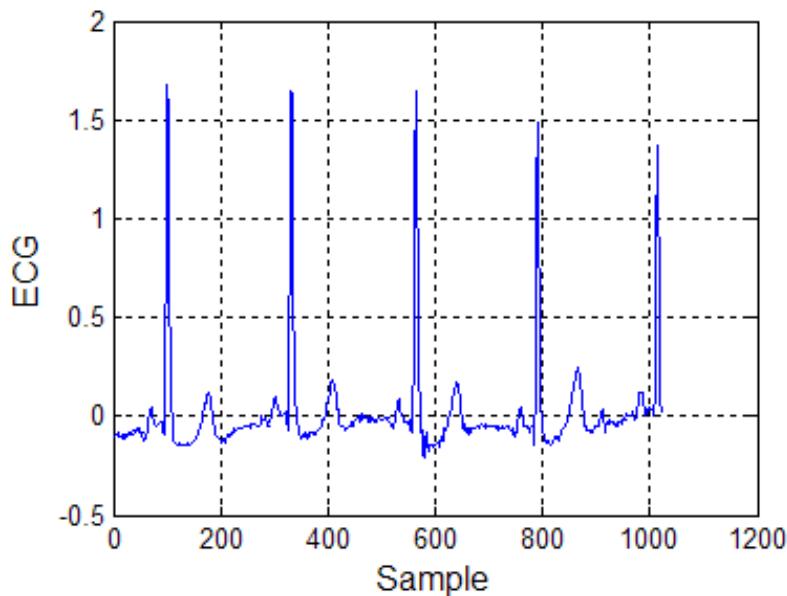
1-D Discrete Cosine Transform



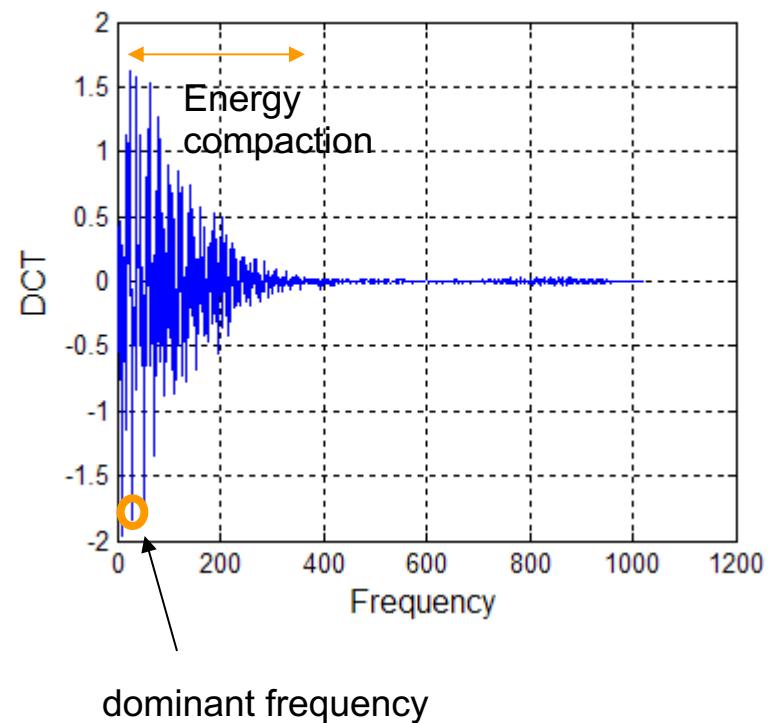
First 8 basis vectors for $N=128$

Basis functions are sinusoids with increasing frequency

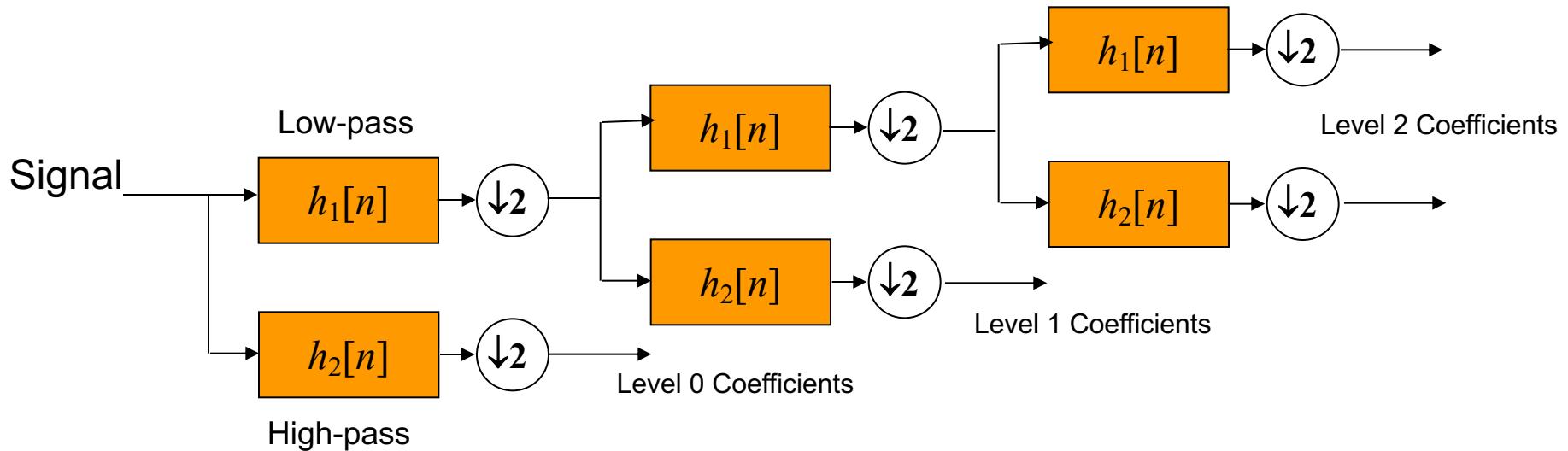
1-D Discrete Cosine Transform



1024 point sampled ECG signal

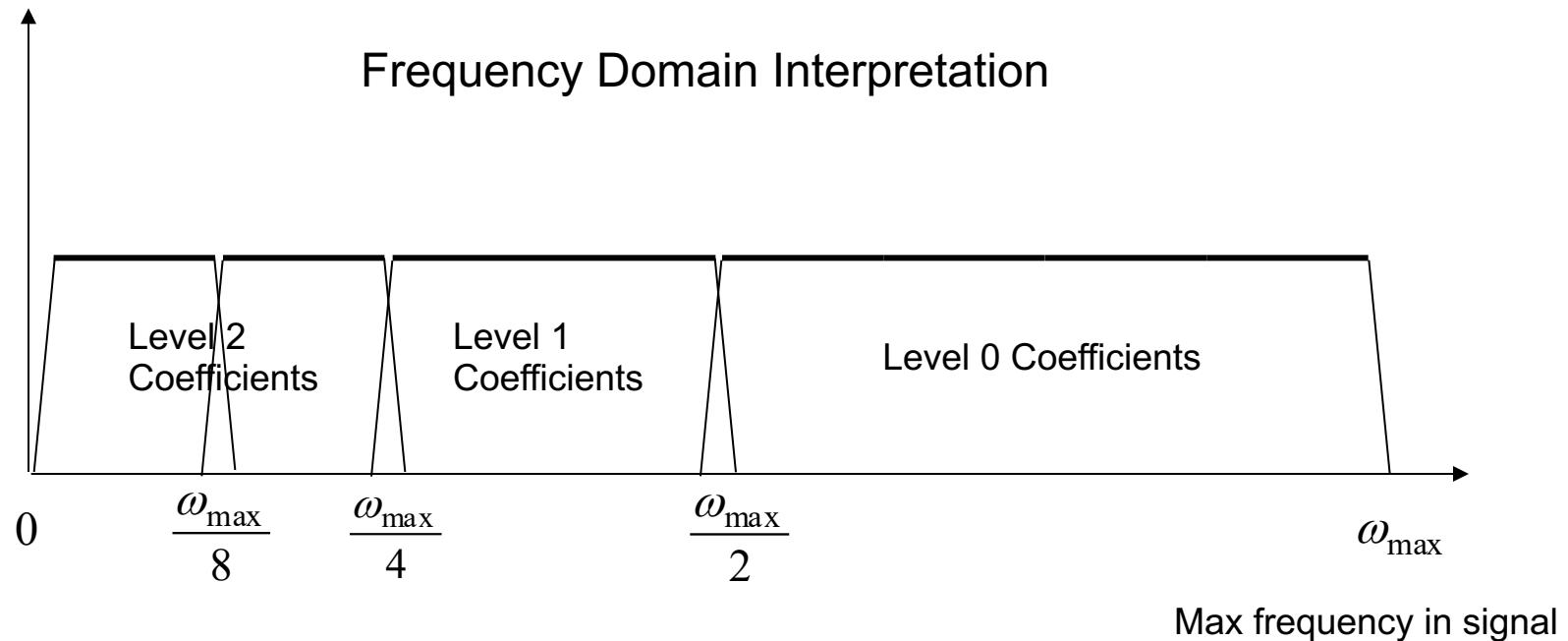


Subband Coding: Wavelet Transform



Recursively apply decomposition to low freq band \Rightarrow increase frequency resolution
Set of filters collectively called – “Filter bank”
Multi-resolution representation of signal – **Scalable Coding**

Subband Coding: Wavelet Transform



Wavelet Transform: Recursively partitions frequency space
Provides time-frequency localization

1-D Haar Wavelet Transform

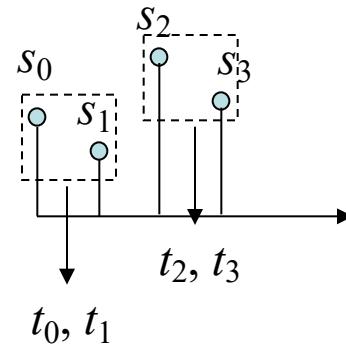
$$\mathbf{U} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

$$\mathbf{U}^H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

$$t_0 = \frac{s_0 + s_1}{\sqrt{2}}$$

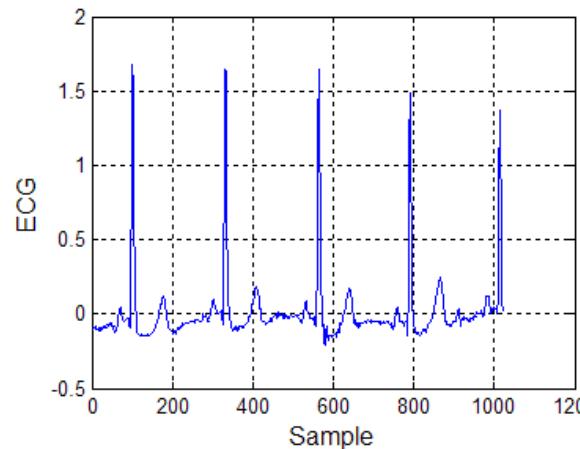
$$t_1 = \frac{s_0 - s_1}{\sqrt{2}}$$

Normalized sum
and difference

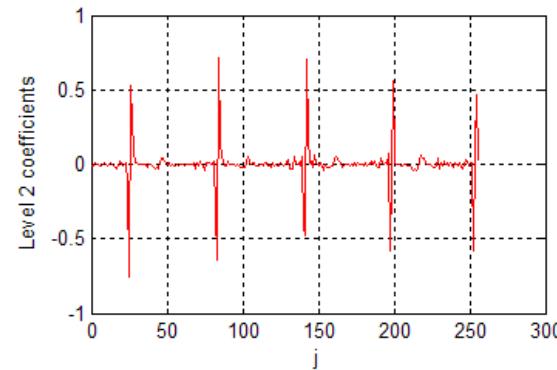
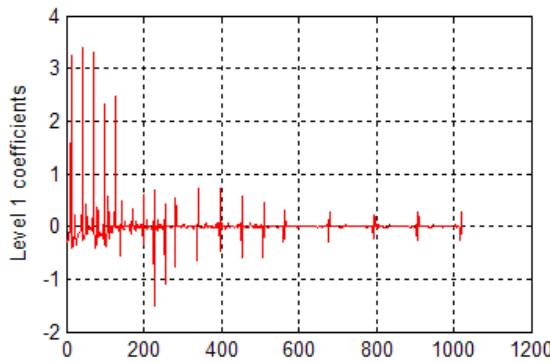


Widnowing Based
Implementation

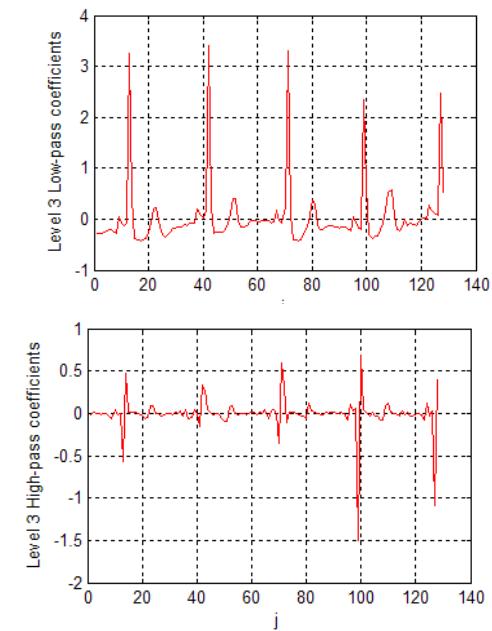
1-D Haar Wavelet Transform



Original
Signal



Level 2 Coefficients

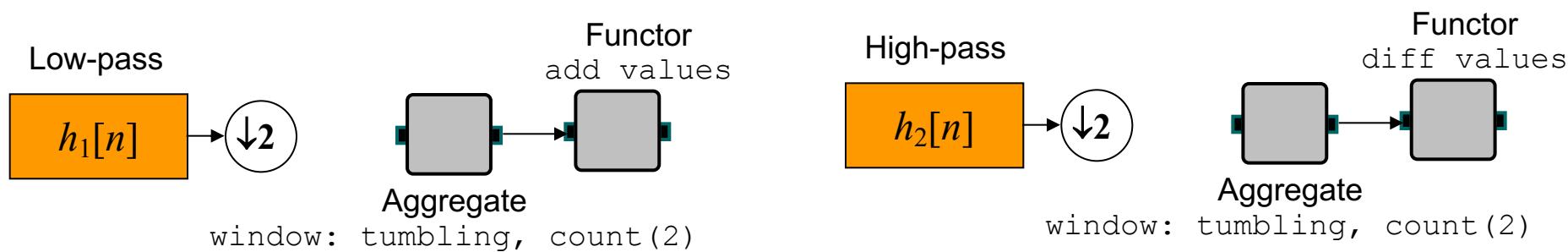
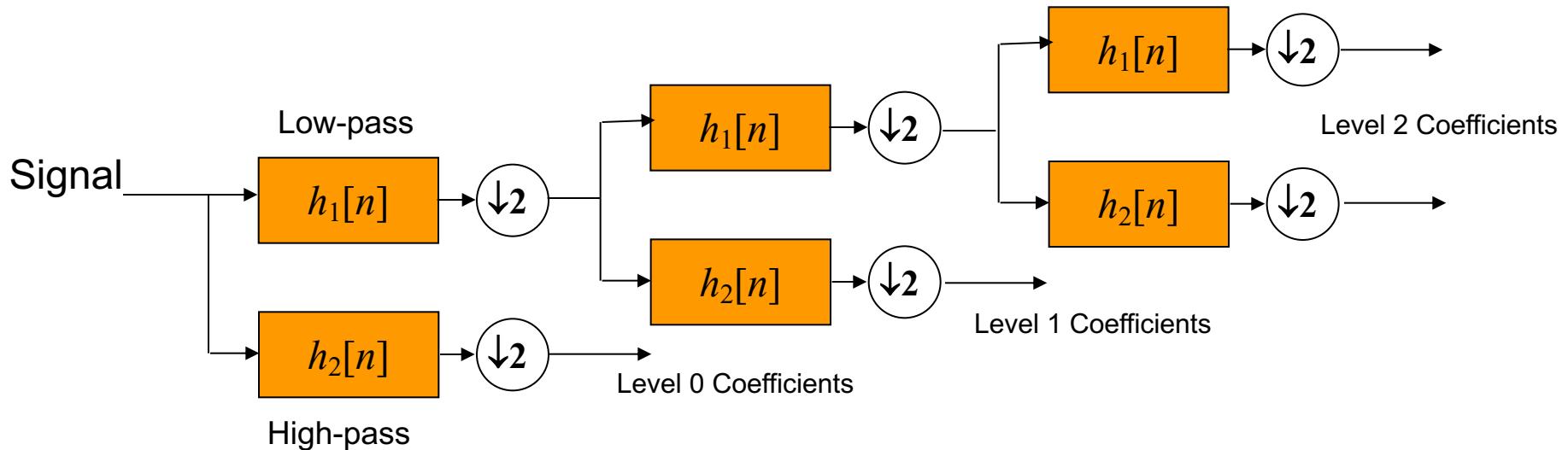


Level 3 Coefficients

Haar Wavelet Transform
How does this relate to compressibility?

Implementing the Temporal Haar Transform

Applied window by window: For a given window, with a fixed size



Implementing the Temporal Haar Transform

- Fixed structure
 - if window size fixed, and known apriori can construct topology at compile time
- What can we do if window size unknown apriori?
 - Construct a transform matrix on the fly
 - Consider a 2 level Haar Transform. *Can we build a single transform matrix for this?*

$$U = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

$$\Phi = \frac{1}{2} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ \sqrt{2} & \sqrt{2} & 0 & 0 \\ 0 & 0 & \sqrt{2} & -\sqrt{2} \end{bmatrix}$$

Implementing the Haar Transform

- Temporal transform with unknown window size
 - Aggregate tuples for the window
 - Construct the transform matrix
 - Apply across the tuples (or tuple attributes)
- Within tuple transform
 - Can be applied tuple by tuple
 - Purely streaming – with transform matrix approach
 - Do not need any aggregation

From 1-D to 2-D Transforms

2-D Transform: Across tuples and across attributes within tuple

$$\Lambda_{N \times N} \begin{bmatrix} x_0(t_0) & x_0(t_i) & x_0(t_{Q-1}) \\ \vdots & \vdots & \vdots \\ x_k(t_0) & x_k(t_i) & x_k(t_{Q-1}) \\ \vdots & \vdots & \vdots \\ x_{N-1}(t_0) & x_{N-1}(t_i) & x_{N-1}(t_{Q-1}) \\ \vdots & & \end{bmatrix} \Lambda_{Q \times Q}^T$$

Within tuple transform Temporal transform

Collect tuples into one matrix $\mathbf{X}_{N \times Q}$

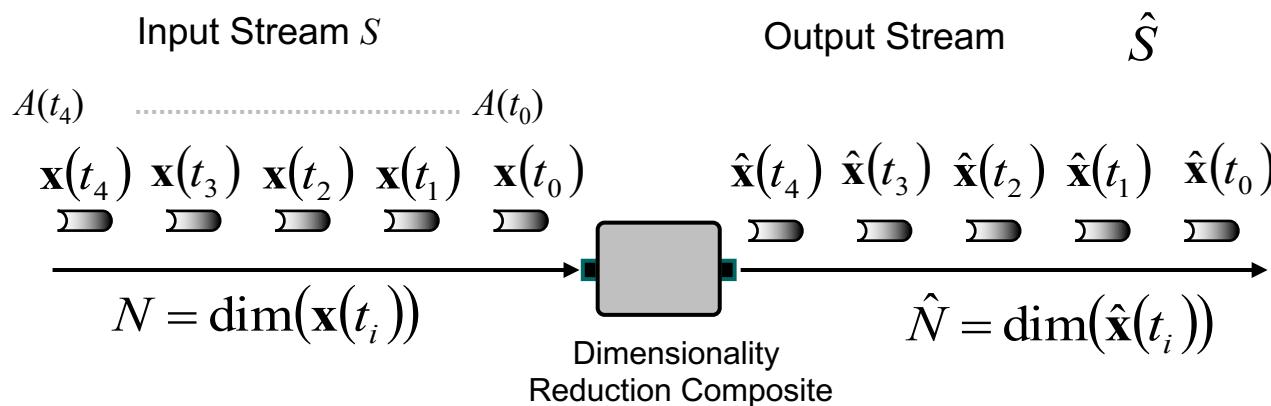
Assume Separability

Transforms: Summary

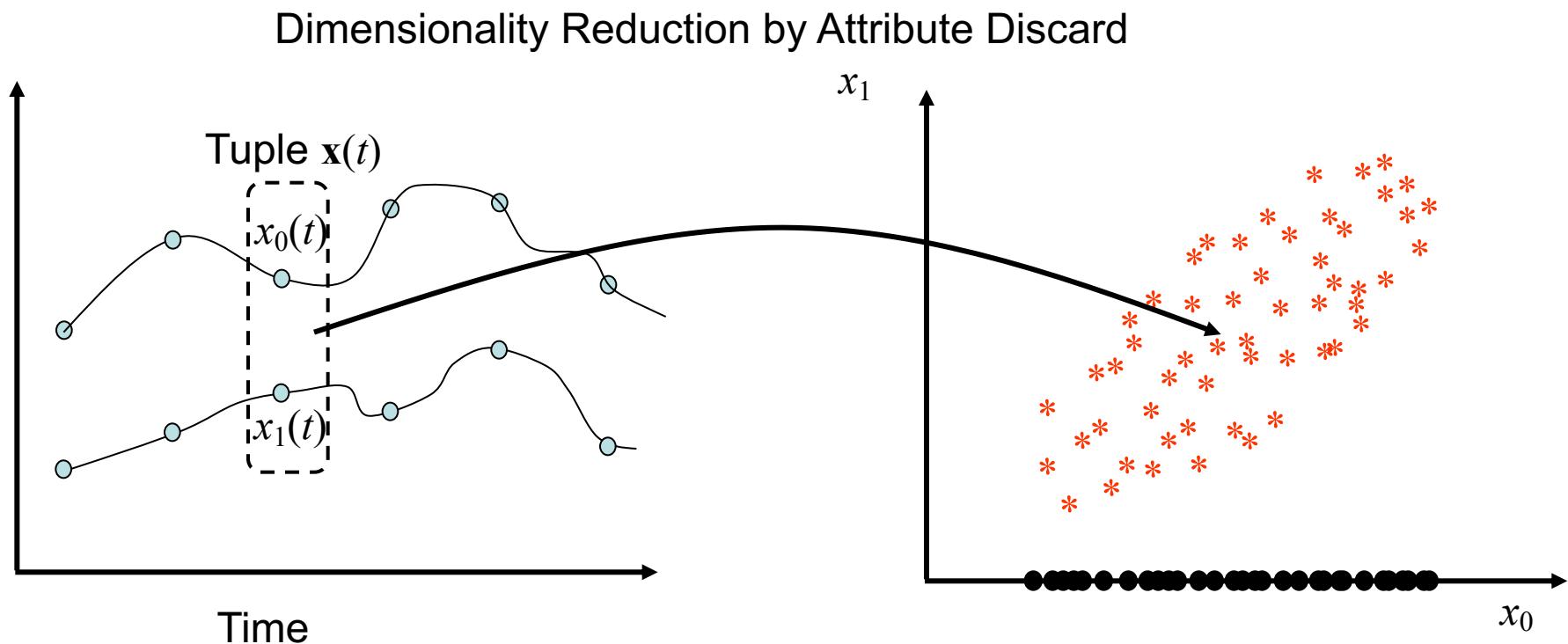
- Capture time-frequency properties of the signal
 - Allow different features to be extracted
- Allow approximating signal with smaller number of coefficients
 - Energy compaction
 - Link to dimensionality reduction and sketches
- Linear transforms
 - Provide windowed computation
 - Provide reversible representations

Dimensionality Reduction

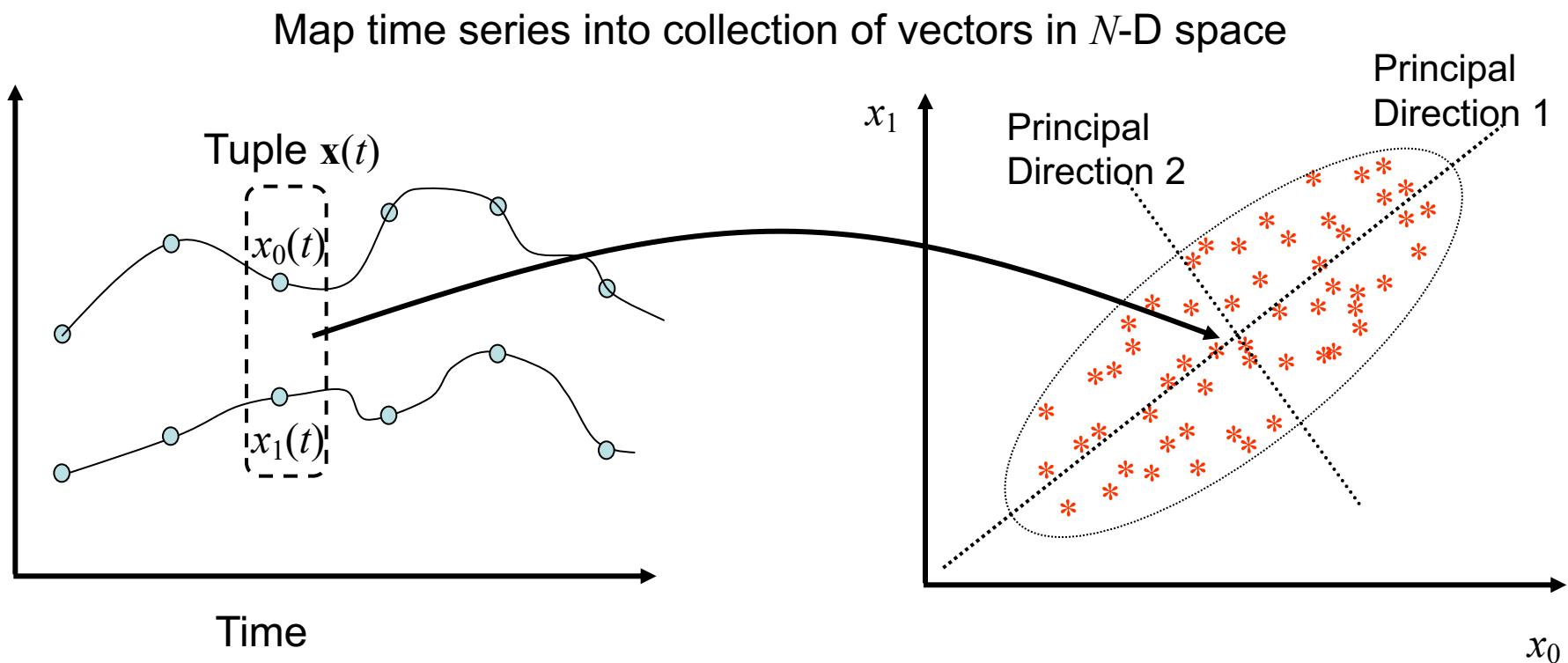
- Applied within tuple
- Reduce number of attributes within tuple
 - From N to \hat{N}
- For numeric tuple
 - Appropriately transform the data into domain where attributes can be discarded
 - See link with energy compacting transforms



Dimensionality Reduction 2-D Example



Dimensionality Reduction: 2-D Example



PCA identifies principal directions of variation in signal

Refresh: PCA and Karhunen-Loewe Transform

- Special kind of unitary transforms
 - Basis vectors are eigenvectors of covariance matrix
 - Decorrelates the dimensions
- Provides guaranteed smallest average squared reconstruction error
 - When only $K (< N)$ coefficients are used to approximate signal
 - Consider that we look at Q time steps

$$\boldsymbol{\mu}_S = \frac{1}{Q} \sum_{j=0}^{Q-1} \mathbf{x}(t_j)$$

Sample Mean vector

$$\boldsymbol{\Sigma}_S = \frac{1}{Q} \sum_{j=0}^{Q-1} (\mathbf{x}(t_j) - \boldsymbol{\mu})(\mathbf{x}(t_j) - \boldsymbol{\mu})^T$$

Sample Covariance matrix

PCA and KLT

The basis vectors for KLT are eigenvectors of the covariance matrix of \mathbf{S}

$$\Sigma_S \mathbf{u}_k = \lambda_k \mathbf{u}_k$$

Using the relationship $\Sigma_T = \mathbf{U}^H \Sigma_S \mathbf{U}$ we have

$$\Sigma_T = \begin{bmatrix} \mathbf{u}_0^H \\ \vdots \\ \mathbf{u}_{N-1}^H \end{bmatrix} \Sigma_S [\mathbf{u}_0 \quad \dots \quad \mathbf{u}_{N-1}]$$
$$\Sigma = \begin{bmatrix} \mathbf{u}_0^H \\ \vdots \\ \mathbf{u}_{N-1}^H \end{bmatrix} [\lambda_0 \mathbf{u}_0 \quad \dots \quad \lambda_{N-1} \mathbf{u}_{N-1}] = \begin{bmatrix} \lambda_0 & 0 & \dots & 0 \\ 0 & \lambda_1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \lambda_{N-1} \end{bmatrix}$$

Diagonal Matrix
Decorrelating transform

The eigenvalues are equal to the variance of transformed coefficients $\lambda_k = \sigma_{t_k}^2$

PCA and KLT

- Provides optimal squared error representation
 - Very efficient dimensionality reduction
 - Added de-correlating property
- Hard to compute in practice
 - Need all vectors available
 - Need to compute covariance matrix
 - Need to compute eigenvectors
 - Not directly suited for streaming data
- Incremental PCA algorithms
 - Approximate PCA

SPIRIT Algorithm

- SPIRIT: Streaming Pattern discoveRy in multiple Timeseries
- Transform based dimensionality reduction
 - Online algorithm: Compute the basis vectors (eigenvectors) incrementally instead of doing an eigen-decomposition
 - Can change amount of reduction incrementally
- Used in multiple applications
 - Forecasting
 - Pattern Detection
 - Anomaly Detection

SPIRIT Algorithm

Input dimensions N
Output dimensions \hat{N}

Initialization

Basis vectors
$$\mathbf{u}_k = \begin{bmatrix} \vdots \\ 1 \\ \vdots \end{bmatrix}$$

Energy (eigenvalue)
 $d_k = \varepsilon$
 $0 \leq k \leq \hat{N}-1$

$\mathbf{z} = \mathbf{x}(t)$

New Tuple

for $k=0: \hat{N}-1$

Iterate through the reduced number of dimensions

$\lambda_k = (\mathbf{z})^T \mathbf{u}_k$

Project onto k -th basis vector

Iterative Update

$d_k = (\lambda_k)^2 + \alpha d_k$

Update energy of k -th basis vector

SPIRIT Algorithm

Iterative Update

$$\mathbf{z} = \mathbf{x}(t)$$

for $k=0:\hat{N}-1$

$$\lambda_k = (\mathbf{z})^T \mathbf{u}_k$$

$$d_k = (\lambda_k)^2 + \alpha d_k$$

$$\mathbf{e}_k = \mathbf{z} - \lambda_k \mathbf{u}_k$$

$$\mathbf{u}_k = \mathbf{u}_k + \frac{1}{d_k} \lambda_k \mathbf{e}_k$$

$$\mathbf{z} = \mathbf{z} - \lambda_k \mathbf{u}_k$$

end

submit vector $\begin{bmatrix} \lambda_0 & \dots & \lambda_{\hat{N}-1} \end{bmatrix}$

New Tuple

Iterate through the reduced number of dimensions

Project onto k -th basis vector

Update energy of k -th basis vector

Find approximation error

Update basis vector

Update basis vector

Dimensionality Reduction

- SPIRIT
 - Can produce approximations to PCA
 - Number of vectors can be changed incrementally (based on error measures)
- Other techniques
 - MUSCLES
 - Distributed Dimensionality Reduction
 - SVD based dimensionality reduction

Summary

Techniques	Tuple Types	Windowing
Sketches	Numeric or non-numeric Tuples, Univariate or Multivariate	Not windowed. Can be tumbling window
Quantization	Numeric tuples. Univariate (scalar) or Multivariate (vector)	Tumbling window based
Transforms	Numeric tuples. Univariate (across tuples) or multivariate (within tuples) or both	Not windowed. Temporal transforms typically tumbling window
Dimensionality Reduction	Numeric tuples. Multivariate	Not windowed. Can be tumbling window

Wrapup

- Pre-processing Algorithms
 - Sketches
 - Quantization
 - Transforms
 - Dimensionality Reduction
- Next lecture
 - Stream mining and modeling algorithms
 - Classification
 - Regression

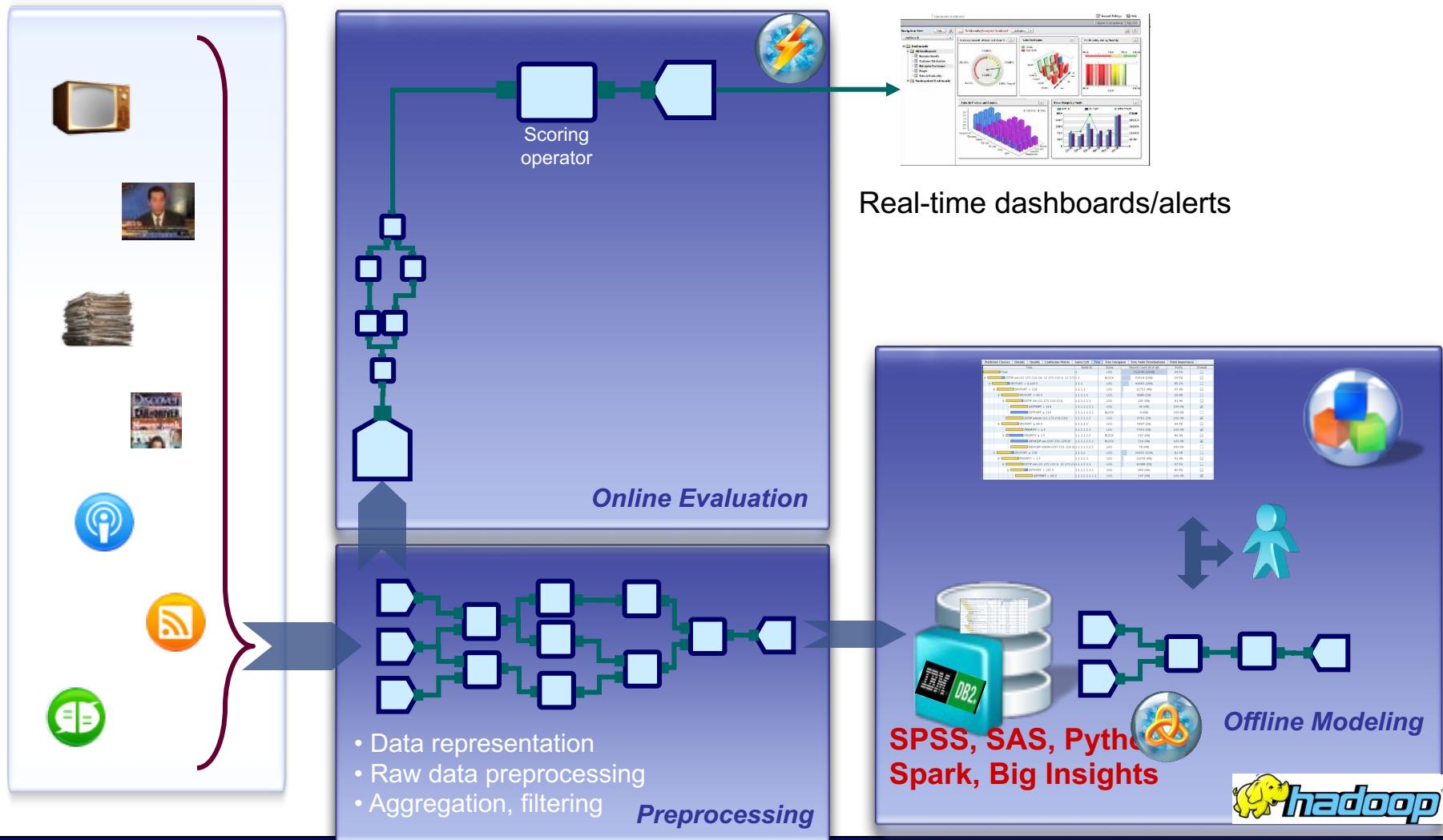
Modeling and Evaluation

- Modeling (model learning)
 - the process of applying knowledge discovery algorithms to the data to extract patterns and trends of interest, resulting in one or more *model*
 - Example: Clustering
 - data: grades of students
 - algorithm: k -means
 - model: k groups with mean grades
 - Usually the model is small compared to the data
- Two kinds
 - Supervised: training data is labeled (e.g., classification, regression)
 - Unsupervised: otherwise (e.g., clustering, frequent pattern mining)
- Various tools and libraries exist for modeling
 - Python, Spark, Matlab, R, SPSS, SAS, Weka

Modeling and Evaluation

- Model scoring (evaluation)
 - using the generated model on new data items, to predict what patterns they match
 - Example: Given a new grade, determine which category it belongs
- Model scoring is usually fast
- Model scoring is naturally a stream processing task

Offline Modeling and Online Evaluation



Offline Modeling and Online Evaluation

- Data Mining Group (DMG) has an XML-based vocabulary for defining models
 - PMML: Predictive model markup language
- Advantage:
 - Can use one system/tool to generate the model and another system to score it

Part of the PMML model

```
1  <MiningSchema>
2    <MiningField usageType="active" name="AGE" importance="0.249"/>
3    <MiningField usageType="active" name="NBR_YEARS_CLI" importance="0.065"/>
4    <MiningField usageType="active" name="AVERAGE_BALANCE" importance="0.18"/>
5    <MiningField usageType="active" name="MARITAL_STATUS" importance="0.024"/>
6    <MiningField usageType="active" name="PROFESSION" importance="0.023"/>
7    <MiningField usageType="active" name="BANKCARD" importance="0.449"/>
8    <MiningField usageType="supplementary" name="GENDER"/>
9    <MiningField usageType="predicted" name="ONLINE_ACCESS"/>
10   </MiningSchema>
```

Offline Modeling and Online Evaluation

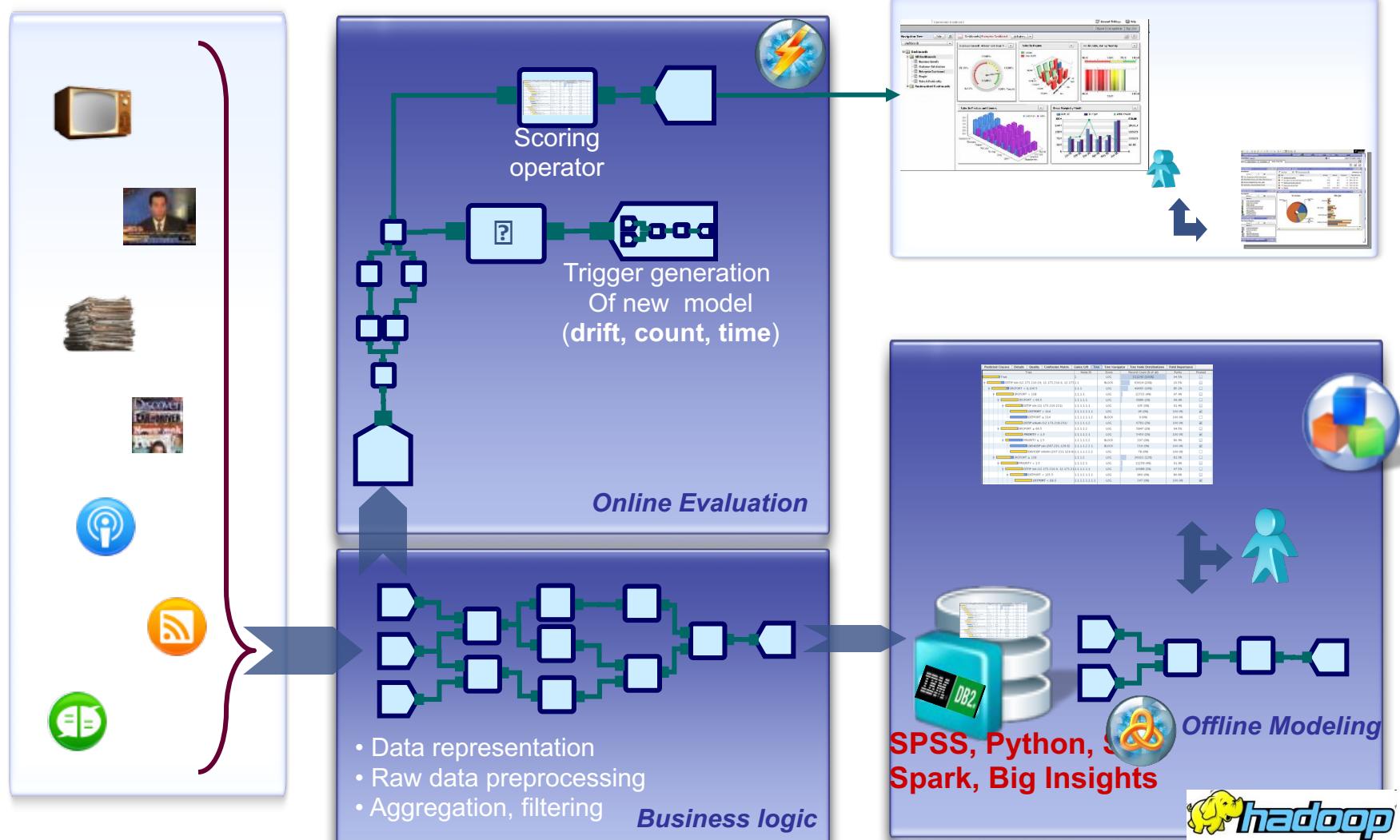
Usage of the PMML model on Streams

```
1 type InType = tuple<int32 age, int32 yearsCli, float64 avgBalance,  
2   rstring maritalStatus, rstring profession, bool hasBankcard>;  
3  
4 stream<InType, tuple<rstring predictedClass, float64 confidence>> Out  
5   = Classification(In)  
6 {  
7   param  
8     model: "bankonlineaccess.pmml"; ← PMML model  
9     age: "AGE";  
10    yearsCli: "NBR_YEARS_CLI";  
11    avgBalance: "AVERAGE_BALANCE";  
12    maritalStatus: "MARITAL_STATUS";  
13    profession: "PROFESSION";  
14    hasBankcard: "BANKCARD";  
15 }
```

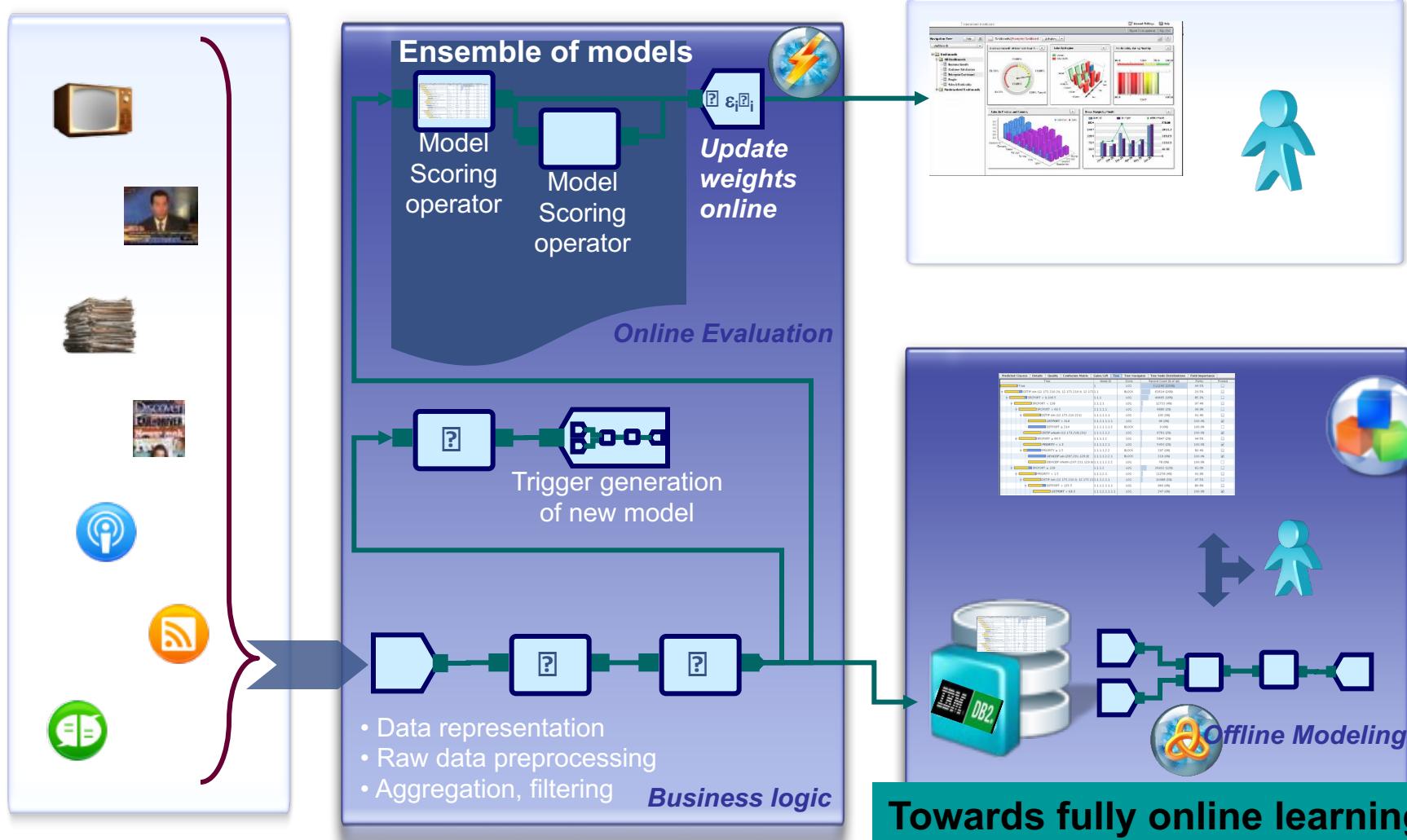
From Offline to Online Modeling

- Offline modeling and online scoring
 - Can be made adaptive by model re-learning and live model switch-over
 - What is the ideal period? Costly to adapt to short term changes.
- Online modeling and online scoring
 - The streaming way of doing things
 - Quick adaptation to changes in the patterns
 - Not always possible to come up with streaming algorithms
- Is there something in between?
 - Online Ensembles

From Offline to Online Analysis



From Offline to Online Analysis



Reading

- Book – Chapter 10: Preprocessing Algorithms
- Other Reading
 - Sampling and Summarization: Data Streams: Models and Algorithms (Chapter 2, Chapter 9, Chapter 12)
 - Sampling (Uniform, Non-uniform, threshold based)
 - Signal Processing Text
 - Reservoir Sampling
 - <http://www.cs.umd.edu/~samir/498/vitter.pdf>
 - Linear Transforms
 - Signal Processing Text (Alan V. Oppenheim, Ronald W. Schafer, John R. Buck : Discrete-Time Signal Processing)
 - Quantization: Signal Processing Text
 - Task specific Quantization for Speaker Verification
 - H. Tseng et al, “Quantization for Adapted GMM-based speaker verification”, ICASSP 2006.
 - Moment Preserving Quantization
 - E. Delp, M. Saenz, and P. Salama. "Block Truncation Coding (BTC)," The Handbook of Image and Video Processing. Academic Press, 2000.