Conjugating data mood and tenses:
Simple past, infinite present, fast continuous, simpler imperative, conditional future perfect
Simple past - Traditional Databases

- Database Management System (DBMS):
  - Data relatively static but queries dynamic

- **Queries**
  - **DBMS**
  - **Results**
  - Persistent relations
    - Random access
    - Low update rate
    - Unbounded disk storage
  - One-time queries
    - Finite query result
    - Queries exploit (static) indices

- **Index**
- **Data**
Simple past – Small Data

- Small data
- OnLine Transaction Processing (OLTP) & OnLine Analytic Processing (OLAP)
  - end of one-size fits all
- Transactions store data, moved to warehouse for reporting & analysis: e.g., client orders
- RDBMS difficult to scale
  - Tx processing has substantial overhead for ACID properties, OLAP bottlenecked by ingestion (indexing)
Infinite Present - MapReduce

- Explosive growth of Data Volume (Big Data), e.g. clickstreams
- Dynamic resource availability (Cloud Computing)
- MapReduce
  - much simpler data and processing model
  - log everything, process in parallel (infinitely scalable)
Infinite Present - MapReduce problems

- Costs: data still grows faster than storage and processing capacity

- Latency: everything is on disks, slow iteration (e.g., PageRank, Logistic Regression, Gradient descent, etc.)

- Batch processing model: stale results (e.g., add 1 link)
Infinite Present - Spark

- Raise of in-memory processing: Spark
  - intermediate results can be stored in memory: fast iteration

```scala
messages = textFile(...).filter(_.startsWith("ERROR"))
  .map(_.split(\t')(2))
```

- Can you go even lower latency?
Fast Continuous - Stream Processing Systems

- **Fast Data** systems require quick responses (Velocity): fraud detection, high frequency trading, ad-prediction, etc...

- Event stream processing
  - process one item at a time in real-time, state stored in memory, e.g., online machine learning

- **Transient streams**
  - Sequential access
  - Potentially high rate
  - Bounded main memory

- **Continuous queries**
  - Produce time-dependant result stream
  - Indexing?
Fast Continuous – State in Stream Processing

- Data flow graph with processing operators
  - Apache S4, Twitter Storm, Nokia Dempsy, ...
Fast Continuous - Challenges

- Consider streaming recommender system (collaborative filtering)
  - Input streams: user activities (item purchases, page views, clicks etc)
  - Operators: maintain recommendation matrices
  - Output streams: recommendations for users

- Processing needs to keep up with varying input rate
  - acquire resources dynamically

- Fault-tolerance
  - cannot restart processing from the start if there is a failure
Fast Continuous - Parallelising Stream Processing

- Exploit data parallelism for scaling out
Fast Continuous: Elasticity and Fault tolerance

- **Elasticity**
  - Provisioning for workload peaks unnecessarily conservative

- **Failure tolerance**
  - Failure is a common occurrence
  - Active fault-tolerance requires 2x resources
  - Passive fault-tolerance leads to long recovery times
SEEP: Stream Processing System [SIGMOD’13]

- **Dynamic scale out:**
  - increase resources when workload peaks occur
  - need to partition the state and distribute across operators

- **Hybrid fault-tolerance:**
  - low resource overhead with fast recovery
  - checkpoint state periodically and reprocess a small subset of recent data

Simpler Imperative – dataflow programming

- How do we write data processing queries?
- Common intermediate representation is dataflow models, tedious to use directly
- Functional approach is popular and easy to translate to dataflows
  - e.g. Spark
- Can we do something for poor imperative programmers out there?
Simpler Imperative – Imperative Big Data Processing [USENIX ATC’14]

- Take annotated Java, and translate to a dataflow model

```java
@Partitioned Matrix userItem = new Matrix();
@Partial Matrix coOcc = new Matrix();

void addRating(int user, int item, int rating) {
    userItem.setElement(user, item, rating);
    Vector userRow = userItem.getRow(user);
    for (int i = 0; i < userRow.size(); i++)
        if (userRow.get(i) > 0) {
            int count = coOcc.getElement(item, i);
            coOcc.setElement(item, i, count + 1);
            coOcc.setElement(i, item, count + 1);
        }
}

Vector getRec(int user) {
    Vector userRow = userItem.getRow(user);
    @Partial Vector userRec = @Global coOcc.multiply(
        userRow);
    Vector rec = merge(@Global userRec);
    return rec;
}

Vector merge(@Collection Vector[] allUserRec) {
    Vector rec = new Vector(allUserRec[0].size());
    for (Vector cur : allUserRec)
        for (int i = 0; i < allUserRec.length; i++)
            rec.set(i, cur.get(i) + rec.get(i));
    return rec;
}
```
Simpler Imperative - Imperative Big Data Processing

- Two simple abstractions: partitioned state & partial state

  - Partition state
    - state is partitioned across operators instances

  - Partial state
    - each instance has local state that can be merged when necessary
Conditional Future Perfect – One model to rule them all?

• Recap: Going to the zoo
  ▪ Transaction processing
  ▪ Batch processing: MapReduce
  ▪ Iterative processing: Spark
  ▪ Stream processing: Storm, Seep
  ▪ More: graph processing, deep learning, etc..

• Can a new comprehensive model emerge?
  ▪ Spark can do graph and batch processing, has been extended to "micro-batch" stream processing
  ▪ Microsoft Naiad & our SEEP integrate stream and iterative + batch

• Can we integrate transaction & stream processing?
Conditional Future Perfect – Current Data Processing Architectures

- Online processing
- Near-real-time processing
- Offline processing

User

Web Front End

transactions

events

Apache Kafka

HDFS

Transaction Processing System

Serving Layer

Stream Processing System

Batch Processing

- Different models, complex interoperation, duplicate code, data movement, resource sharing
Conditional Future Perfect – Unified Stream and Transaction processing

- Online processing
  | User |
  | Web Front End |
  | events |
  | transactions |

- Near-real-time processing
  | Apache Kafka |
  | HDFS |

- Offline processing
  | User |
  | Web Front End |

- Serving Layer
- Transaction Processing System
- Stream Processing System
- Batch Processing
- Unified Stream and Transaction Processing System

User access to shared state
Conditional Future Perfect – Unified Stream and Tx processing

- Why now?
  - Renewed interest in in-memory databases, most business core DBs fit in memory!
  - Stream processing are becoming more sophisticated in terms of state management
  - Fault-tolerance approaches are becoming similar: deterministic databases, allow replication/recomputation approaches

- Challenges
  - Unify processing models operations (Tx with non-deterministic ordering) vs events (mostly deterministic)
  - Synchronisation and ordering are key to scalable performance
  - Performance vs generality tradeoffs (80% on 80%?)
Conclusion

• Rapid evolution of data processing systems
  ▪ Transaction processing
  ▪ MapReduce
  ▪ In-Memory Processing
  ▪ Stream processing

• Each with their own strength

• Lack of cohesive view and interaction problems

• It is time to look back and
  ▪ map existing solutions
  ▪ work on integration, synergies, and tradeoffs

• Happy to collaborate on any of the above