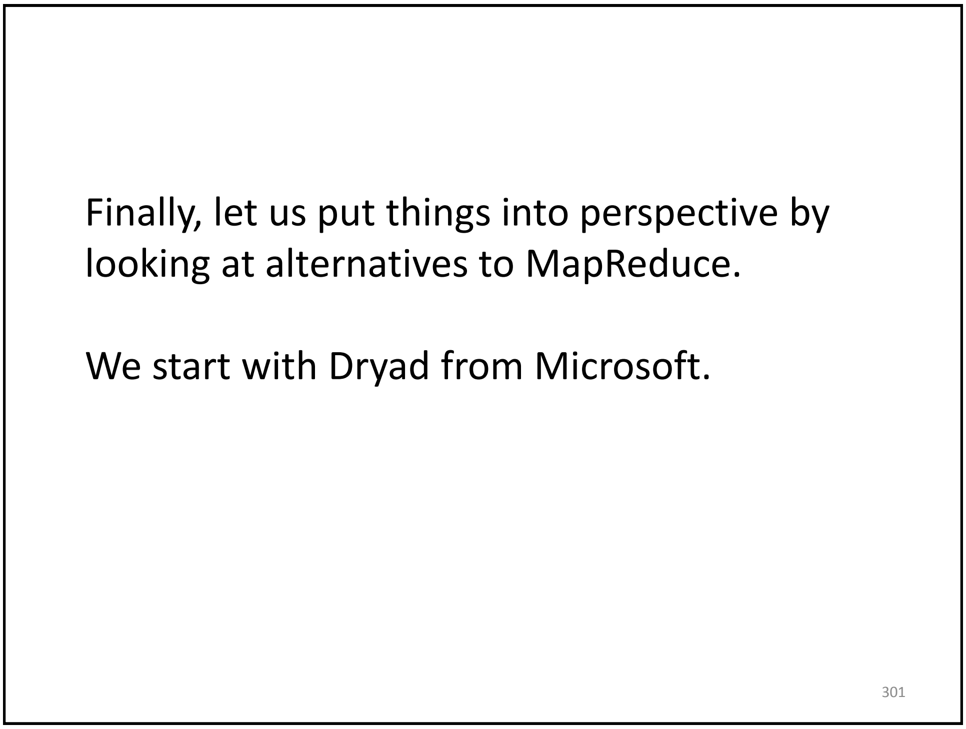
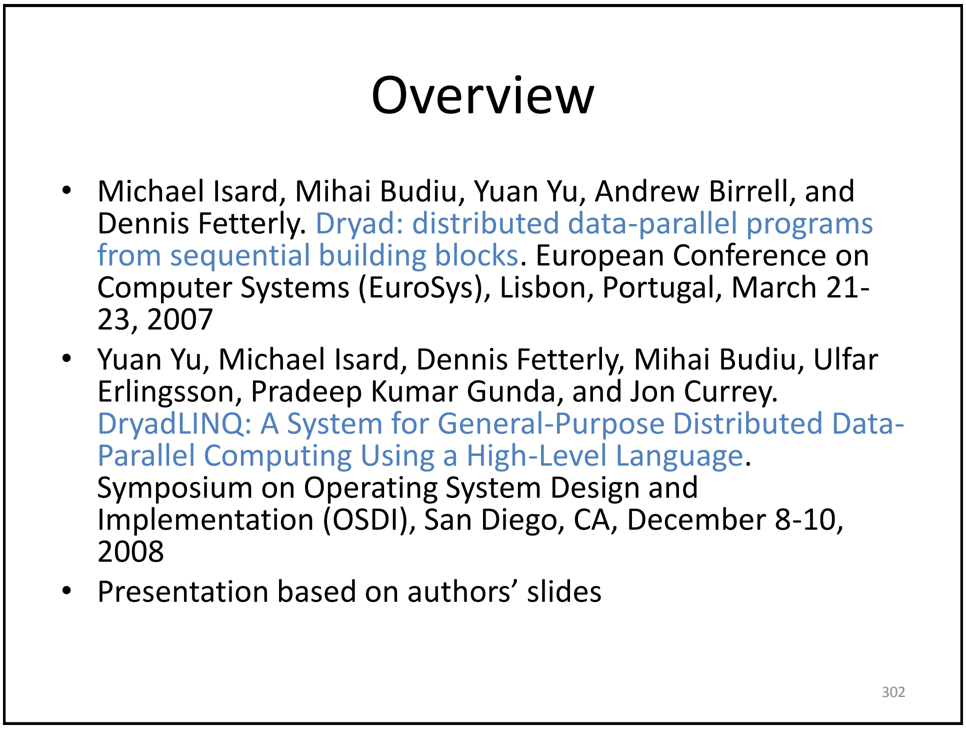


Finally, let us put things into perspective by looking at alternatives to MapReduce.

We start with Dryad from Microsoft.



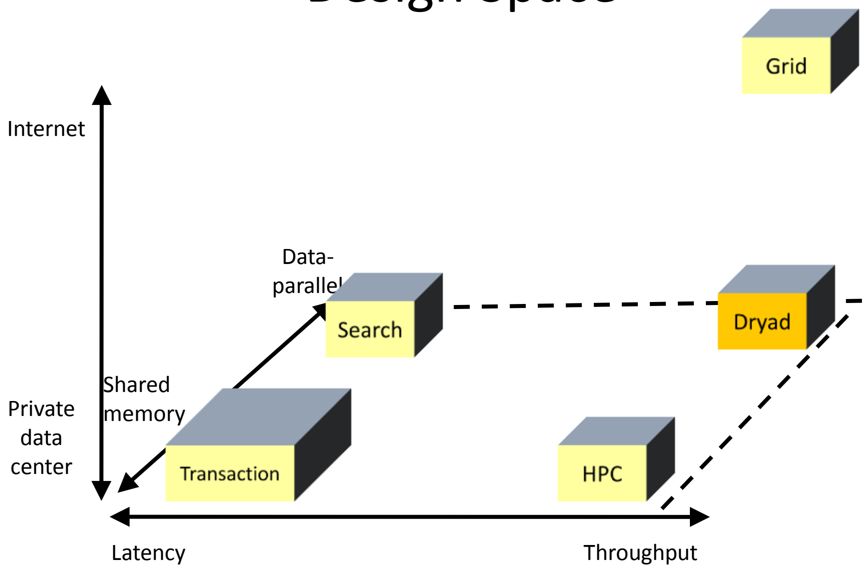
# Overview

- Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. European Conference on Computer Systems (EuroSys), Lisbon, Portugal, March 21-23, 2007
- Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Ulfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. Symposium on Operating System Design and Implementation (OSDI), San Diego, CA, December 8-10, 2008
- Presentation based on authors' slides

# Outline

- Dryad Design
- Implementation
- Policies as Plug-ins
- Building on Dryad

# Design Space



Grid

Internet

Data-parallel

Search

Dryad

Shared memory

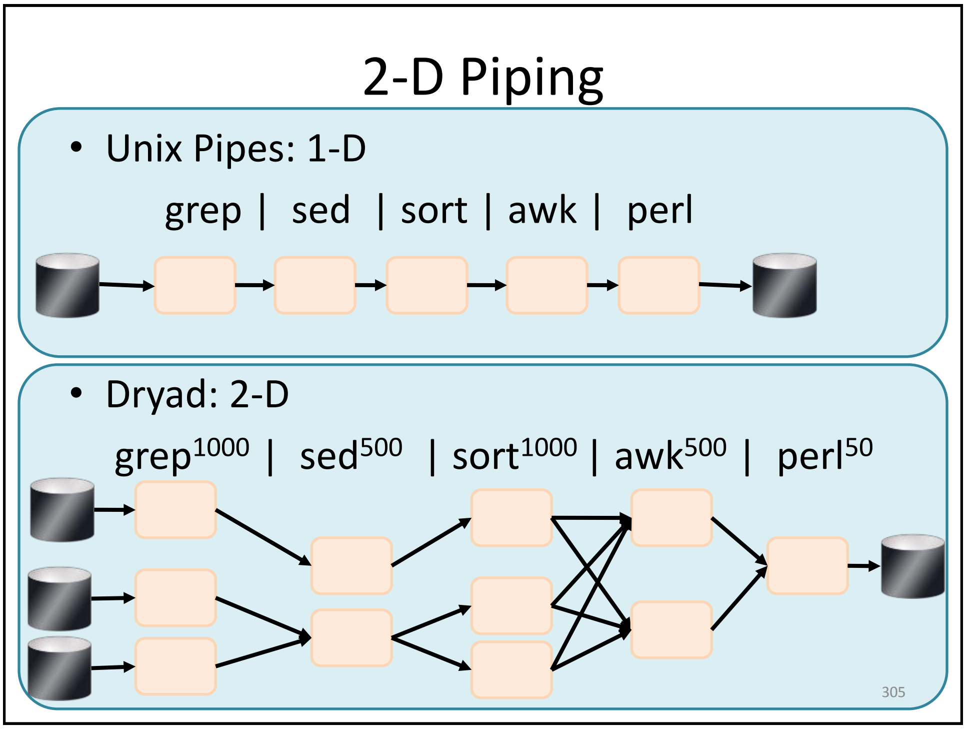
Private data center

Transaction

HPC

Latency

Throughput



## 2-D Piping

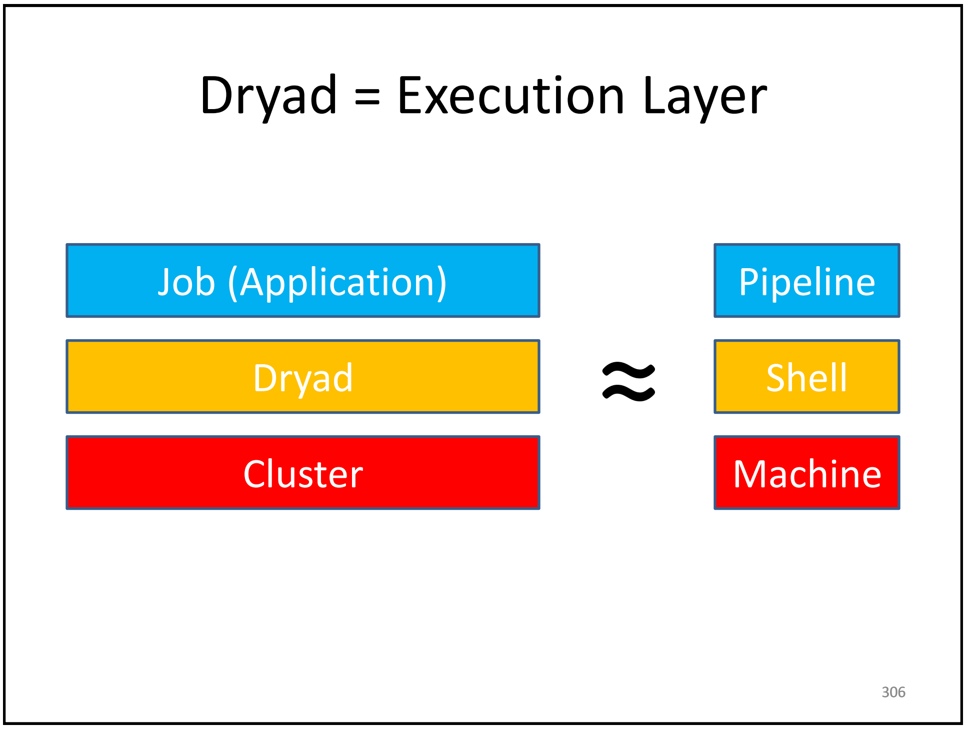
- Unix Pipes: 1-D

  grep | sed | sort | awk | perl

- Dryad: 2-D

  grep$^{1000}$ | sed$^{500}$ | sort$^{1000}$ | awk$^{500}$ | perl$^{50}$

305



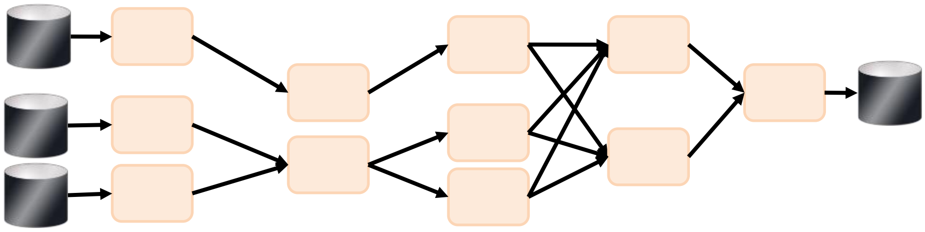
## Dryad = Execution Layer

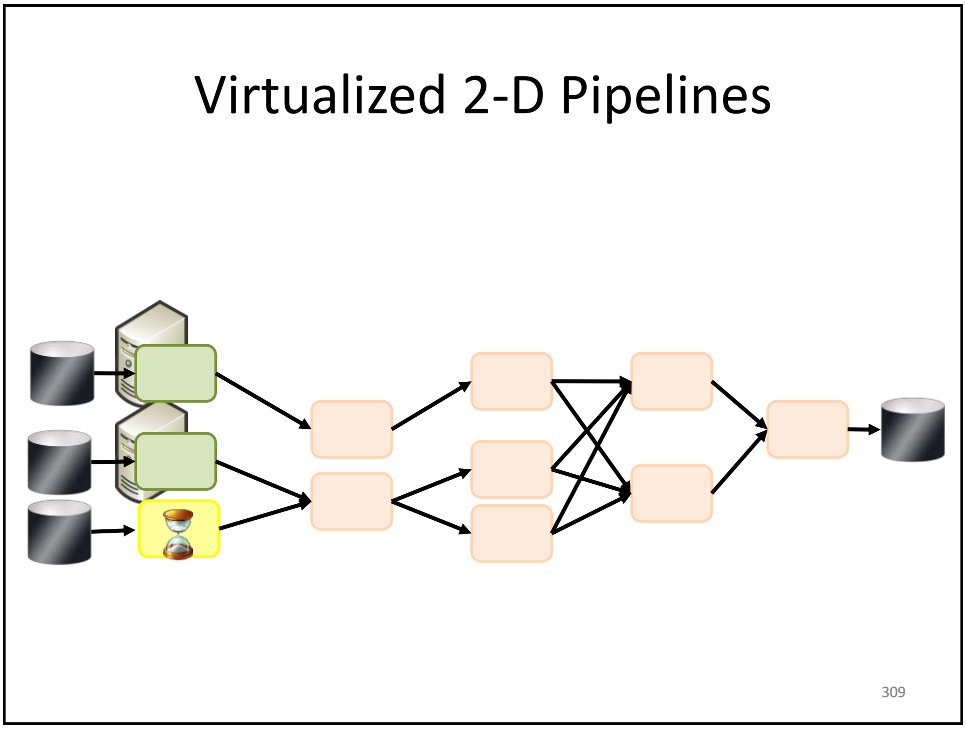
306

# Outline

- Dryad Design
- Implementation
- Policies as Plug-ins
- Building on Dryad

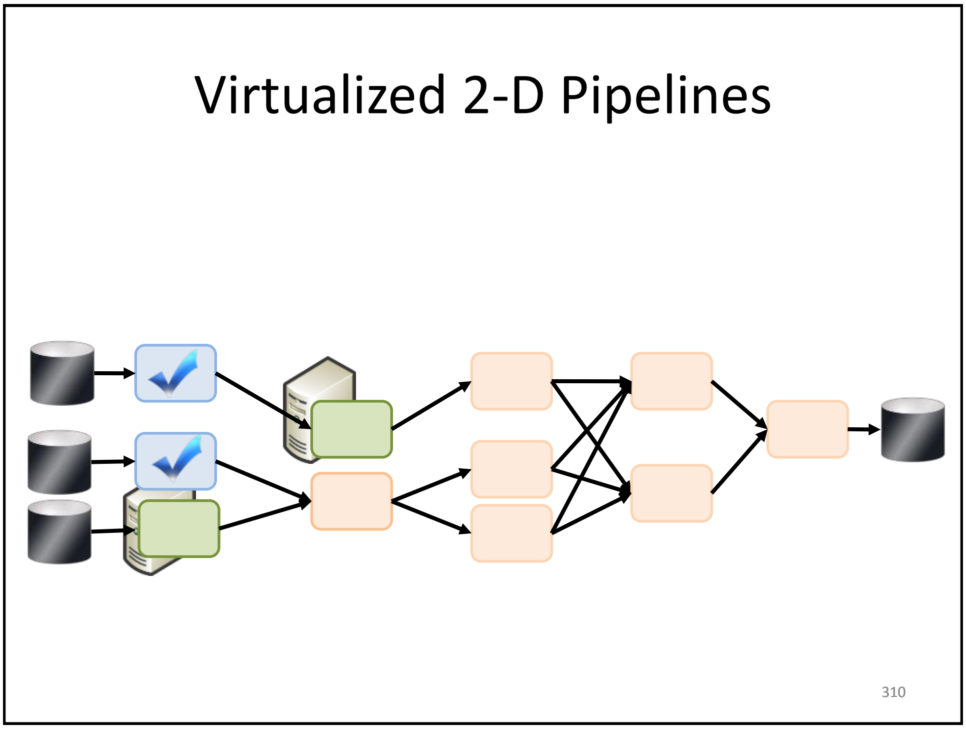
# Virtualized 2-D Pipelines





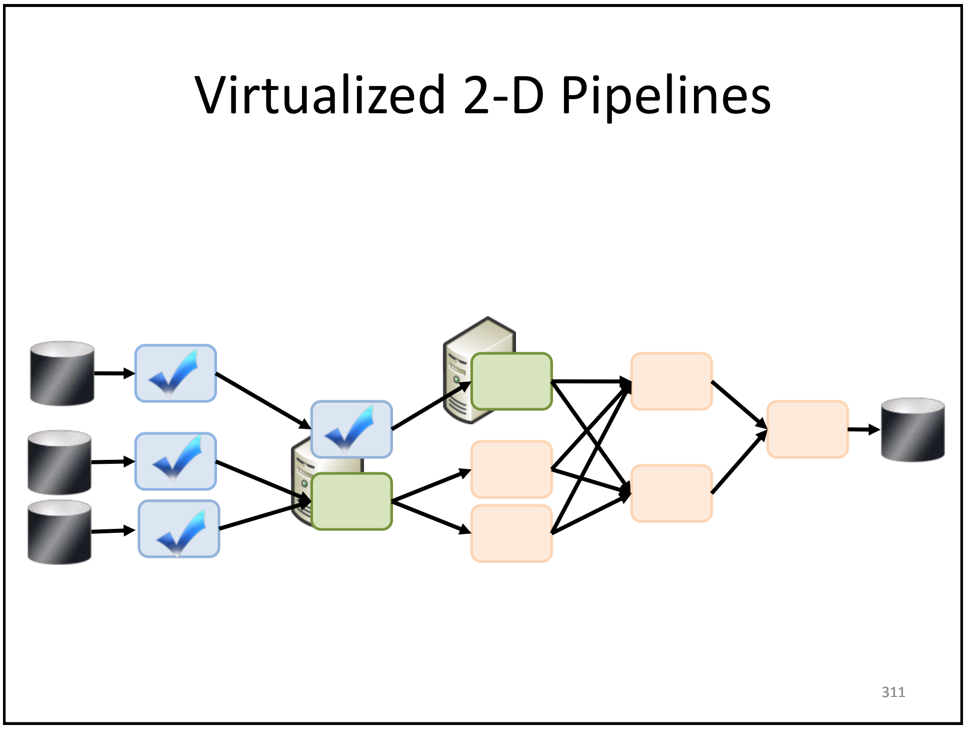
## Virtualized 2-D Pipelines

309

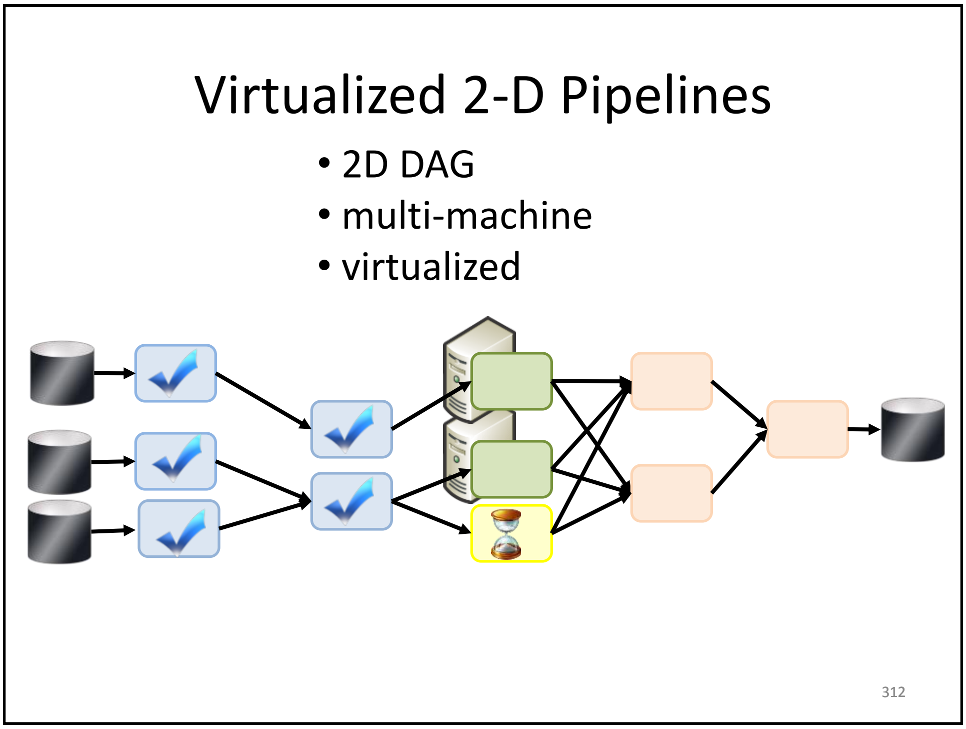


## Virtualized 2-D Pipelines

310

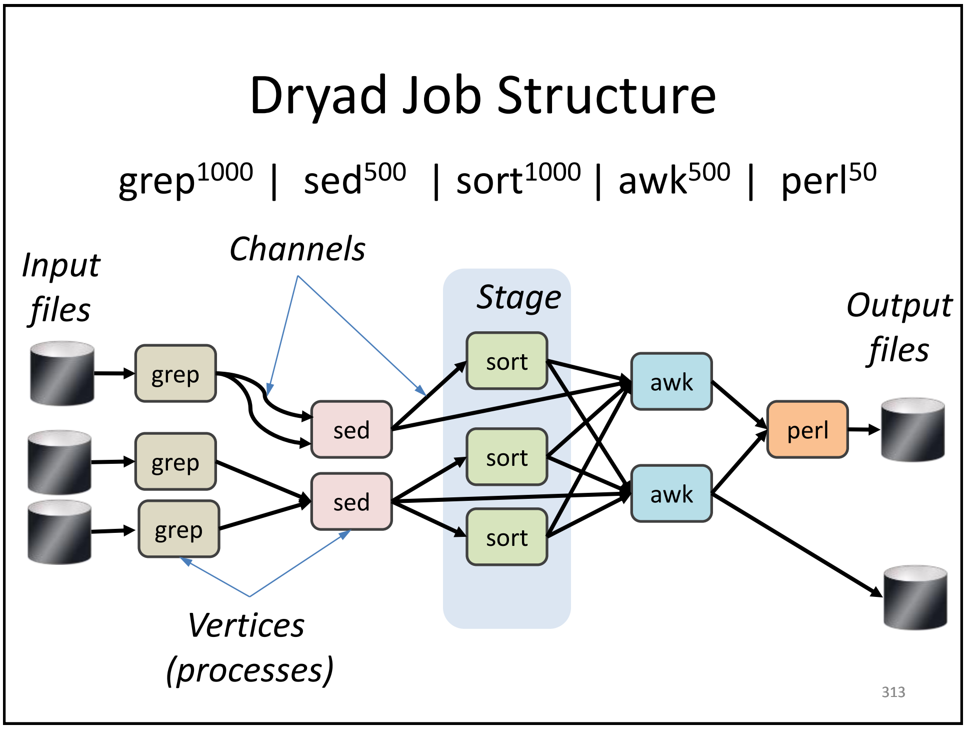


## Virtualized 2-D Pipelines



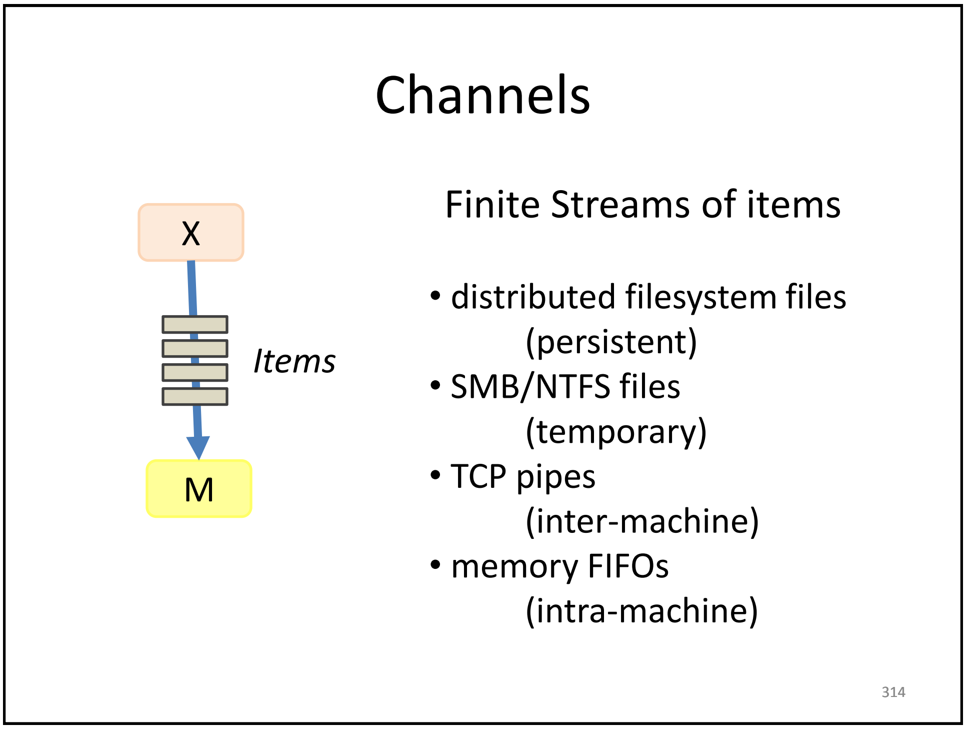
## Virtualized 2-D Pipelines

- 2D DAG
- multi-machine
- virtualized



# Dryad Job Structure

grep$^{1000}$ | sed$^{500}$ | sort$^{1000}$ | awk$^{500}$ | perl$^{50}$

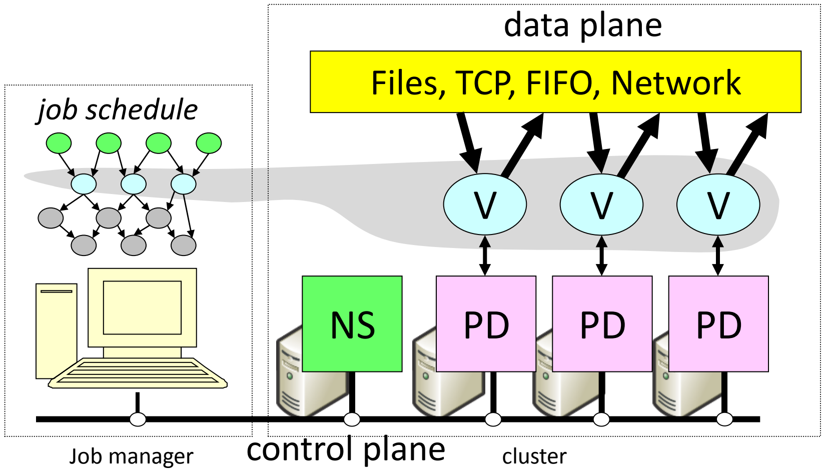


# Channels

Finite Streams of items

- distributed filesystem files
  (persistent)
- SMB/NTFS files
  (temporary)
- TCP pipes
  (inter-machine)
- memory FIFOs
  (intra-machine)

# Architecture



data plane

Files, TCP, FIFO, Network

job schedule

V

V

V

NS

PD

PD

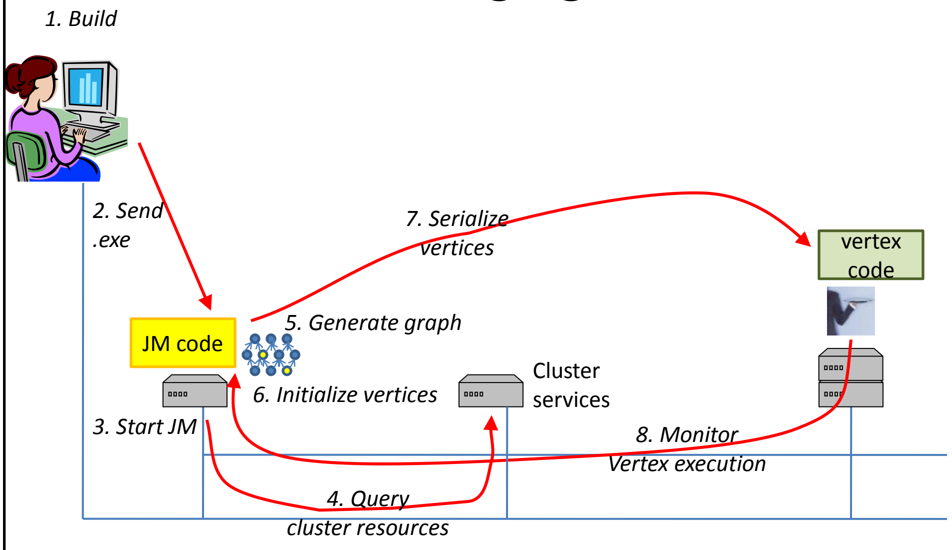
PD

Job manager

control plane

cluster

# Staging



1. Build

2. Send .exe

3. Start JM

JM code

4. Query cluster resources

Cluster services

5. Generate graph

6. Initialize vertices

7. Serialize vertices

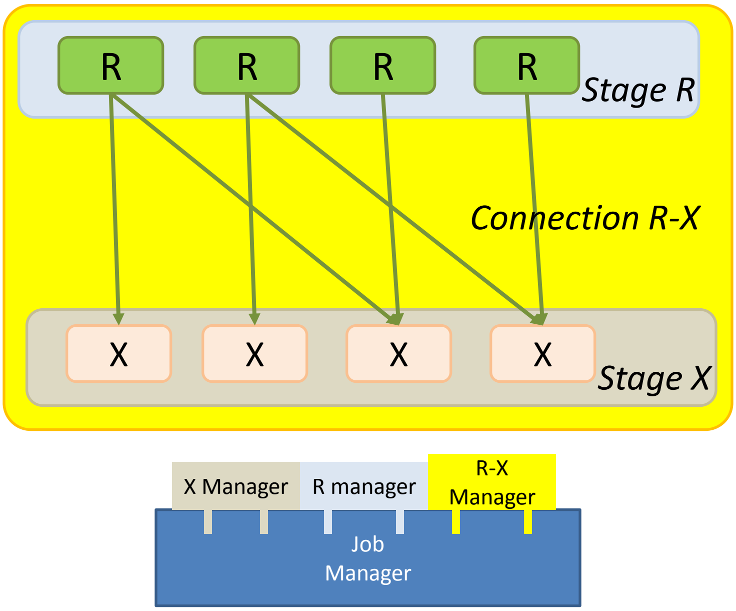
vertex code

8. Monitor Vertex execution

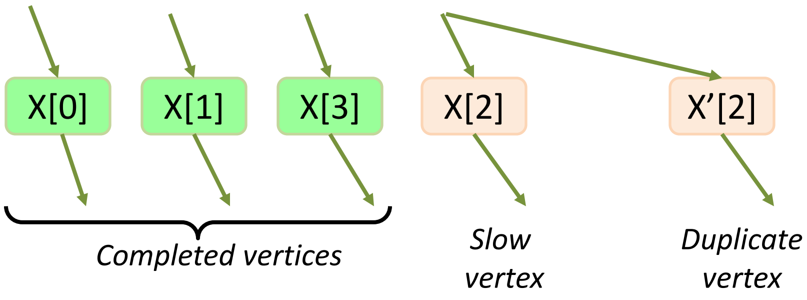
# Outline

- Dryad Design
- Implementation
- Policies  and Resource Management
- Building on Dryad

# Policy Managers



# Duplicate Execution Manager



X[0]
X[1]
X[3]
X[2]
X’[2]

Completed vertices

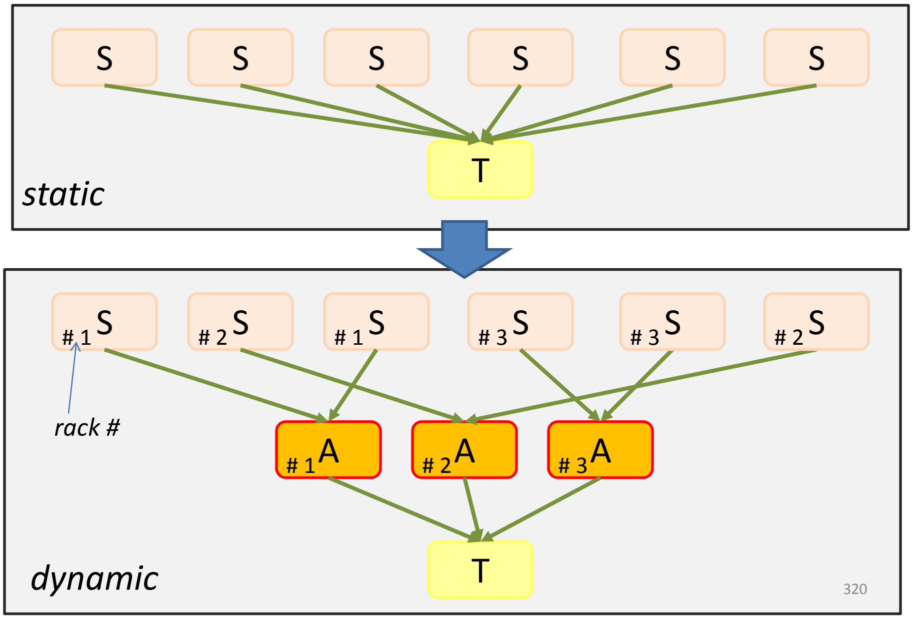
Slow vertex

Duplicate vertex

Duplication Policy = f(running times, data volumes)

319

# Aggregation Manager



S S S S S S
T
static

# 1 S
# 2 S
# 1 S
# 3 S
# 3 S
# 2 S

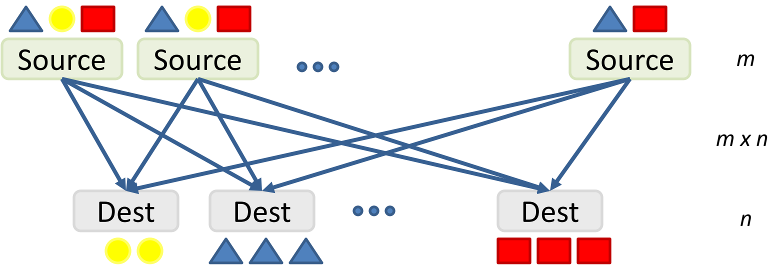
rack #

# 1 A
# 2 A
# 3 A

T
dynamic

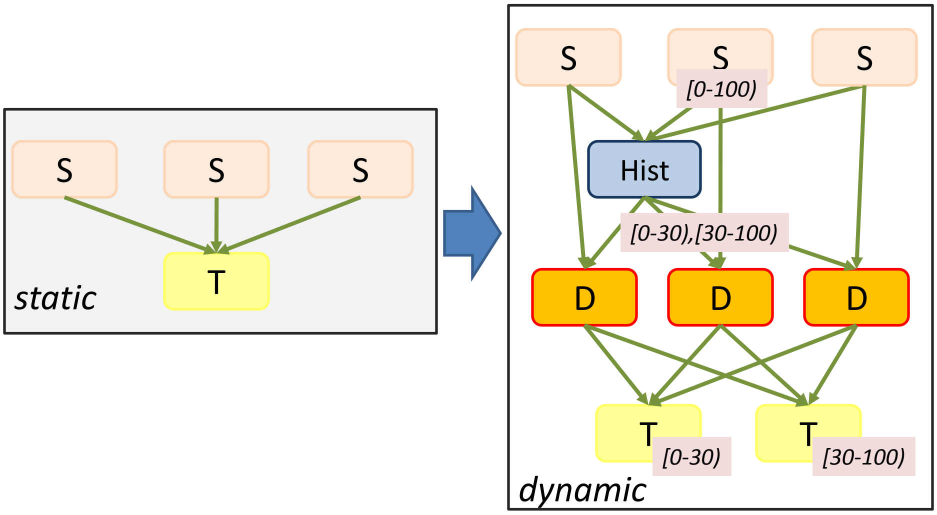
320

# Data Distribution
(Group By)



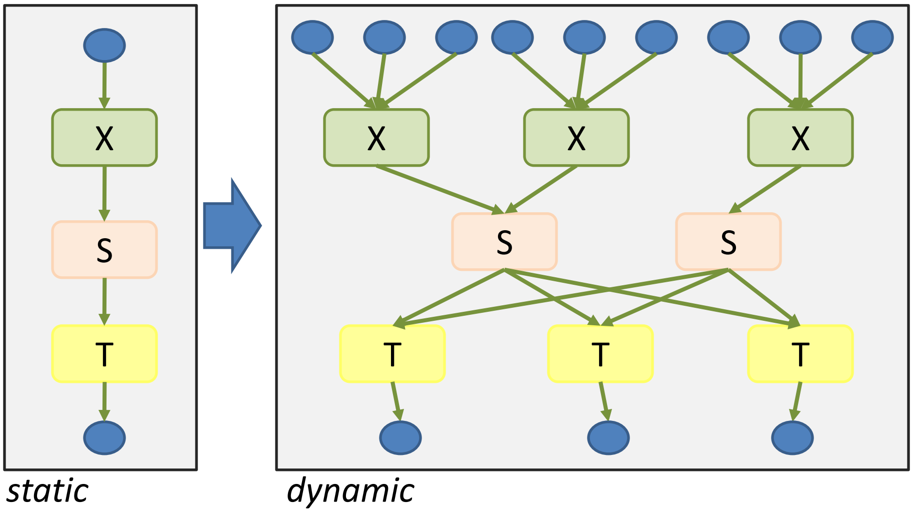
321

# Range-Distribution Manager



322

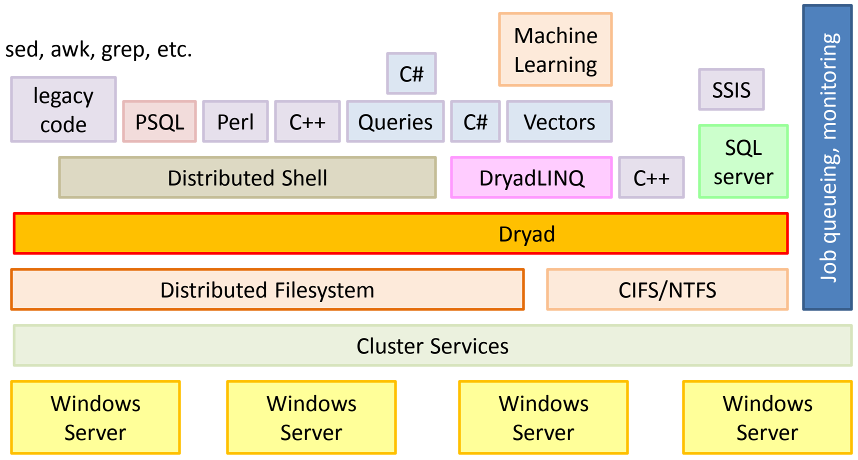
## Goal: Declarative Programming



## Outline

- Dryad Design
- Implementation
- Policies as Plug-ins
- Building on Dryad

# Software Stack



sed, awk, grep, etc.

legacy code | PSQL | Perl | C++ | C# Queries | C# | Machine Learning Vectors | SSIS

Distributed Shell | DryadLINQ | C++ | SQL server

Dryad

Distributed Filesystem | CIFS/NTFS

Cluster Services

Windows Server | Windows Server | Windows Server | Windows Server

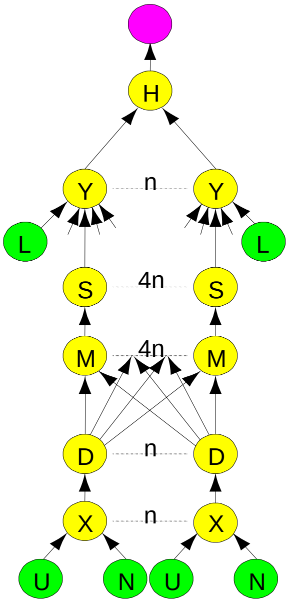
Job queueing, monitoring

# Example Query: Sky Server

- Table photoPrimary
  - All identified astronomical objects (354,254,163 records)
  - ID, color magnitude in 5 bands (u, g, r, i, z)
- Table neighbors
  - For each object, neighbors within 30 arc seconds (2,803,165,372 records)
- Query 18: gravitational lens effect
  - Find all objects that have neighbors whose color is similar to that object

## SkyServer Query 18

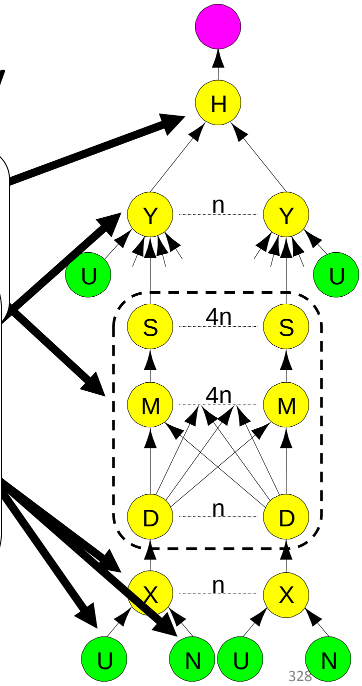
```
select distinct U.ObjID
into results
  from photoPrimary U,
         neighbors N,
         photoPrimary L
where U.ObjID = N.ObjID
   and U.mode = 1
   and L.ObjID = N.NeighborObjID
   and U.ObjID < L.ObjID
   and abs((U.u-U.g)-(L.u-L.g))<0.05
   and abs((U.g-U.r)-(L.g-L.r))<0.05
   and abs((U.r-U.i)-(L.r-L.i))<0.05
   and abs((U.i-U.z)-(L.i-L.z))<0.05
```

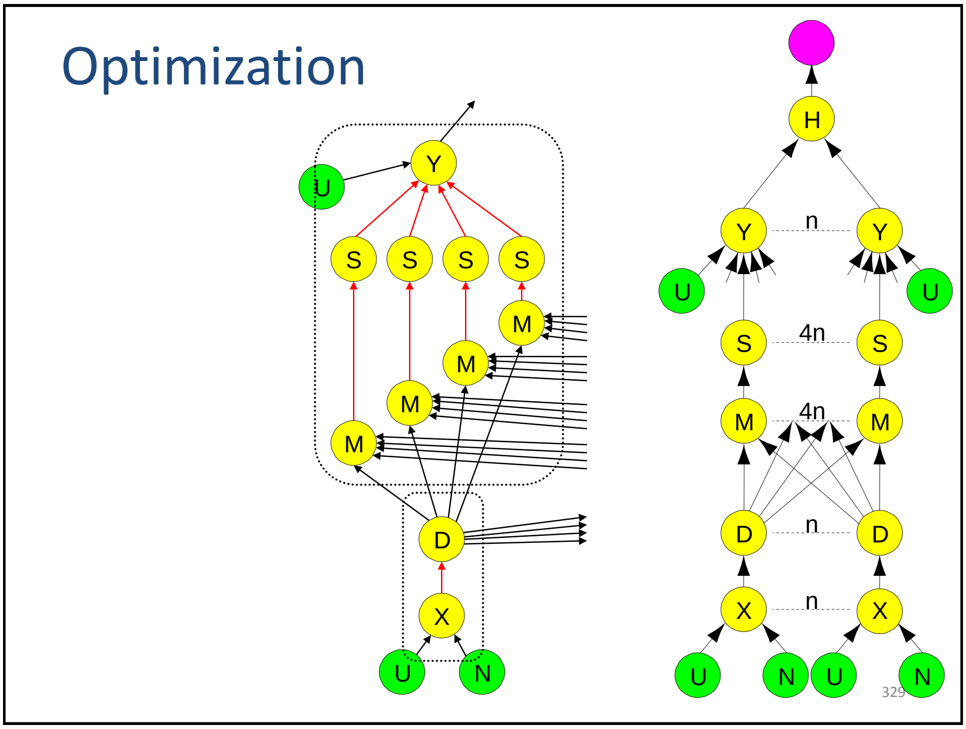


## SkyServer DB query

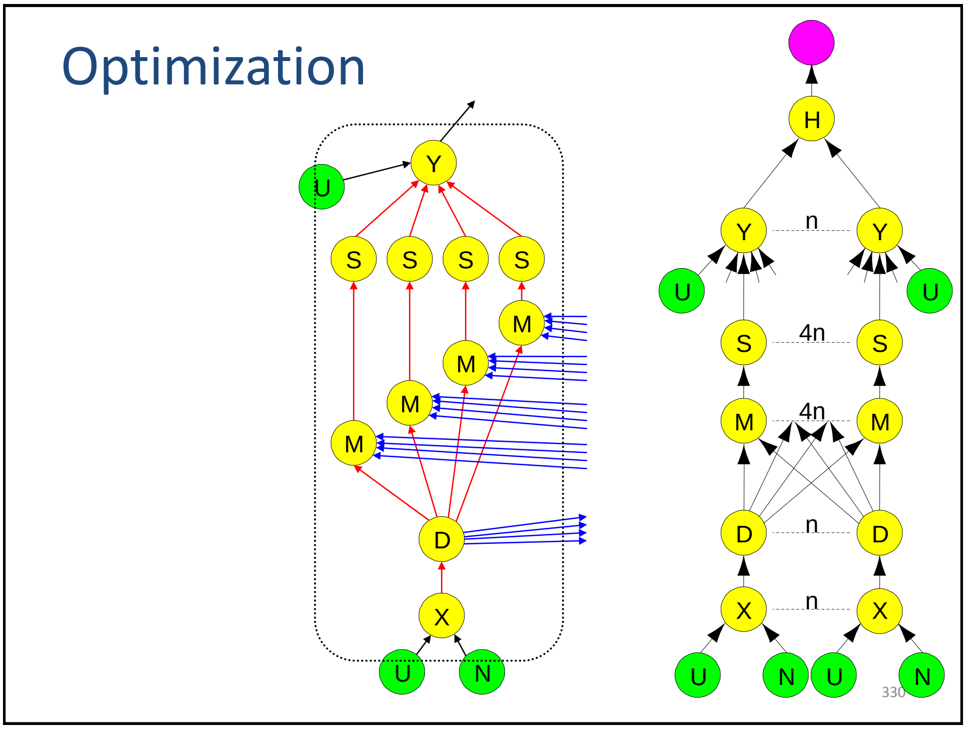
**[distinct]**
**[merge outputs]**

```
select
  u.objid
from u join <temp>
where
  u.objid = <temp>.neighborobjid and
  |u.color - <temp>.color| < d
```



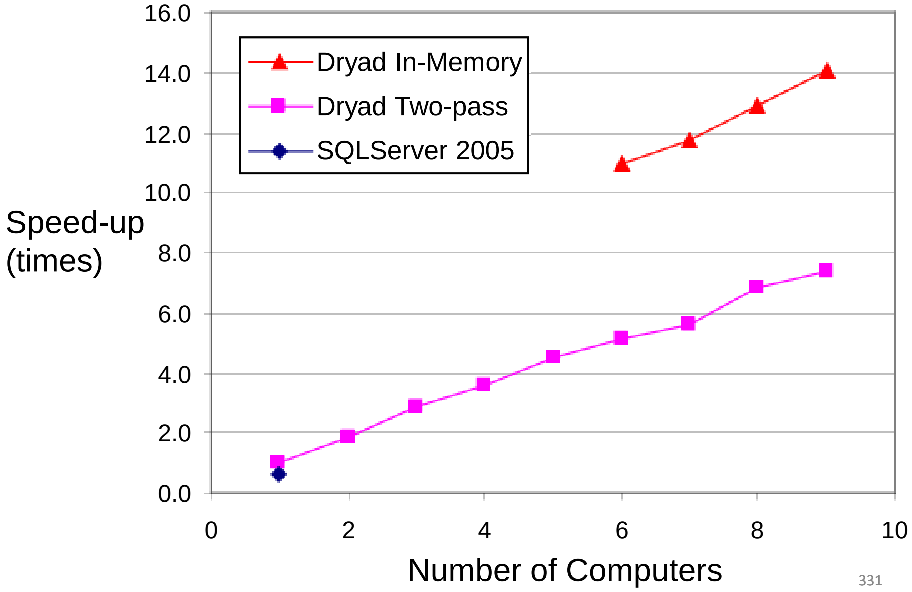


## Optimization



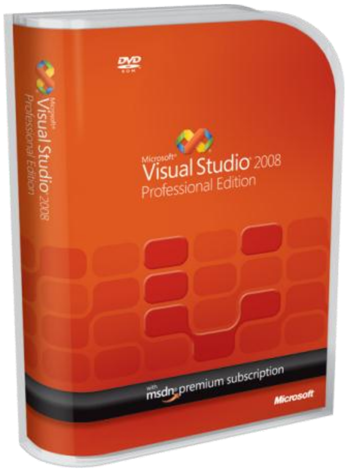
## Optimization

## SkyServer Q18 Performance



## DryadLINQ

- Declarative programming
- Integration with Visual Studio
- Integration with .Net
- Type safety
- Automatic serialization
- Job graph optimizations
  - static
  - dynamic
- Conciseness


Microsoft Visual Studio 2008 Professional Edition
with msdn premium subscription
Microsoft

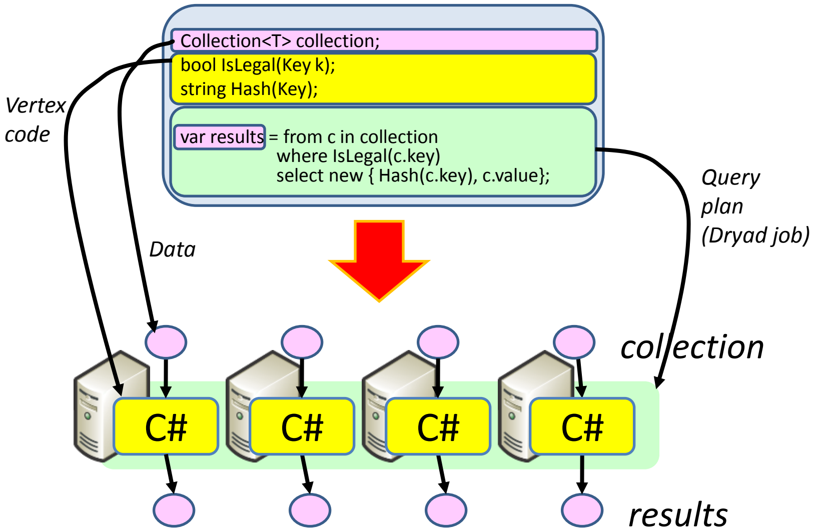
## LINQ

```
Collection<T> collection;
bool IsLegal(Key);
string Hash(Key);

var results = from c in collection
              where IsLegal(c.key)
              select new { Hash(c.key), c.value};
```

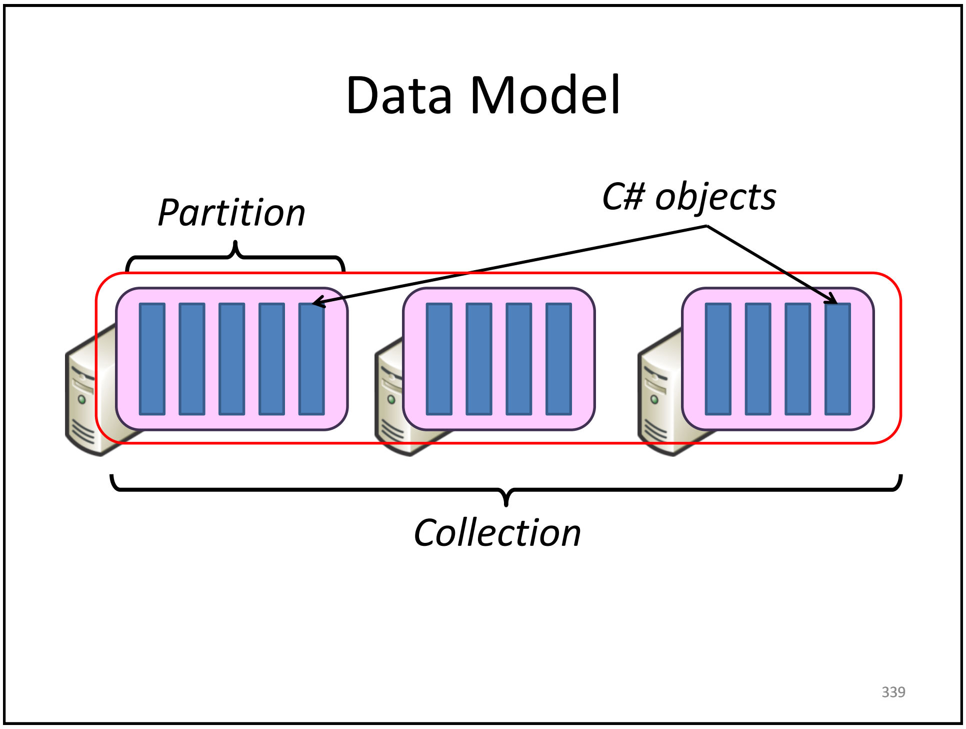
337

## DryadLINQ = LINQ + Dryad



```
Collection<T> collection;
bool IsLegal(Key k);
string Hash(Key);
var results = from c in collection
              where IsLegal(c.key)
              select new { Hash(c.key), c.value};
```

338



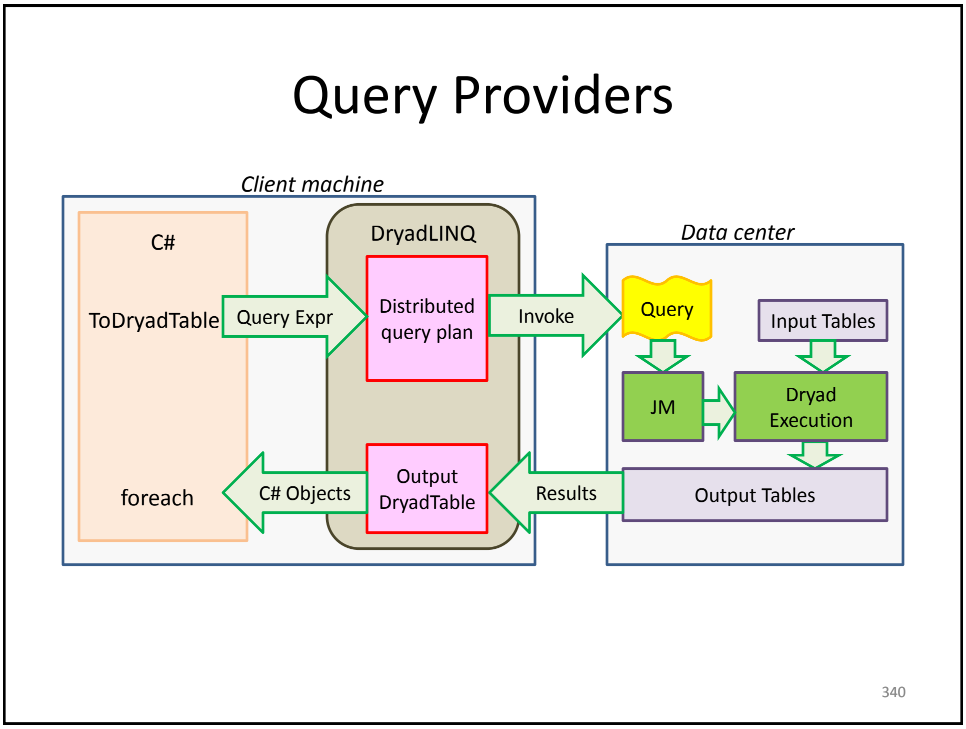
# Data Model

Partition

C# objects

Collection

339



# Query Providers

Client machine

C#

ToDryadTable

foreach

DryadLINQ

Query Expr

Distributed query plan

Output DryadTable

C# Objects

Invoke

Results

Data center

Query

Input Tables

JM

Dryad Execution

Output Tables

340

# Example: Histogram

```
public static IQueryable<Pair> Histogram(
    IQueryable<LineRecord> input, int k)
{
   var words = input.SelectMany(x => x.line.Split(' '));
   var groups = words.GroupBy(x => x);
   var counts = groups.Select(x => new Pair(x.Key, x.Count()));
   var ordered = counts.OrderByDescending(x => x.count);
   var top = ordered.Take(k);
   return top;
}
```

| |
|---|
| “A line of words of wisdom” |
| [“A”, “line”, “of”, “words”, “of”, “wisdom”] |
| [[“A”], [“line”], [“of”, “of”], [“words”], [“wisdom”]] |
| [ {“A”, 1}, {“line”, 1}, {“of”, 2}, {“words”, 1}, {“wisdom”, 1}] |
| [{“of”, 2}, {“A”, 1}, {“line”, 1}, {“words”, 1}, {“wisdom”, 1}] |
| [{“of”, 2}, {“A”, 1}, {“line”, 1}] |

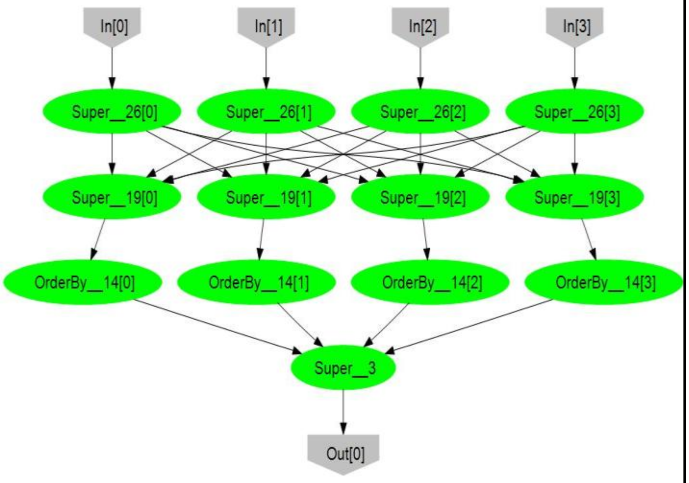
# Histogram Plan

SelectMany
HashDistribute

Merge
GroupBy
Select

OrderByDescending
Take

MergeSort
Take

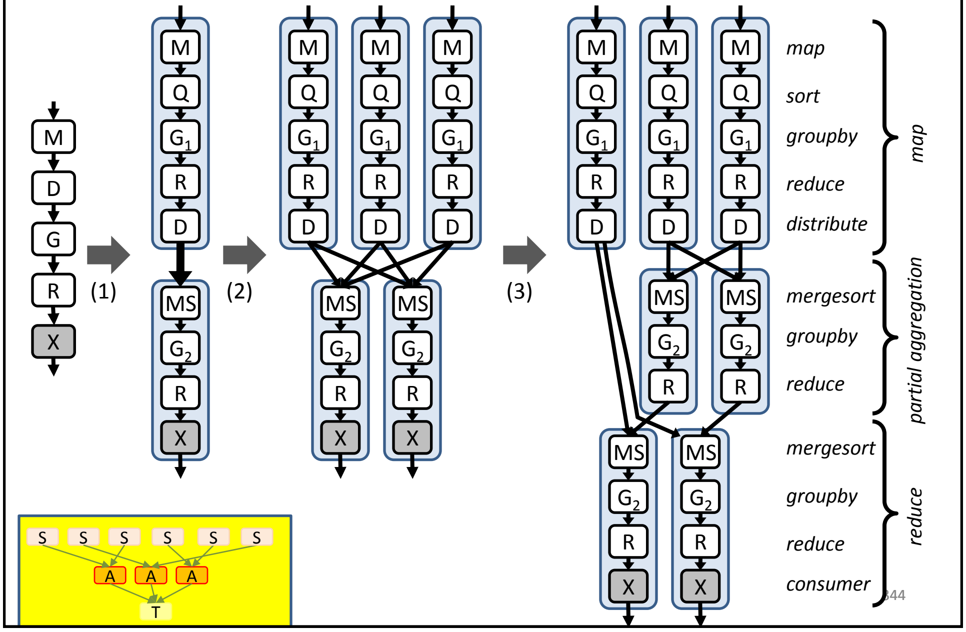


# Map-Reduce in DryadLINQ

```
public static IQueryable<S> MapReduce<T,M,K,S>(
    this IQueryable<T> input,
    Expression<Func<T, IEnumerable<M>>> mapper,
    Expression<Func<M,K>> keySelector,
    Expression<Func<IGrouping<K,M>,S>> reducer)
{
    var map = input.SelectMany(mapper);
    var group = map.GroupBy(keySelector);
    var result = group.Select(reducer);
    return result;
}
```

343

# Map-Reduce Plan

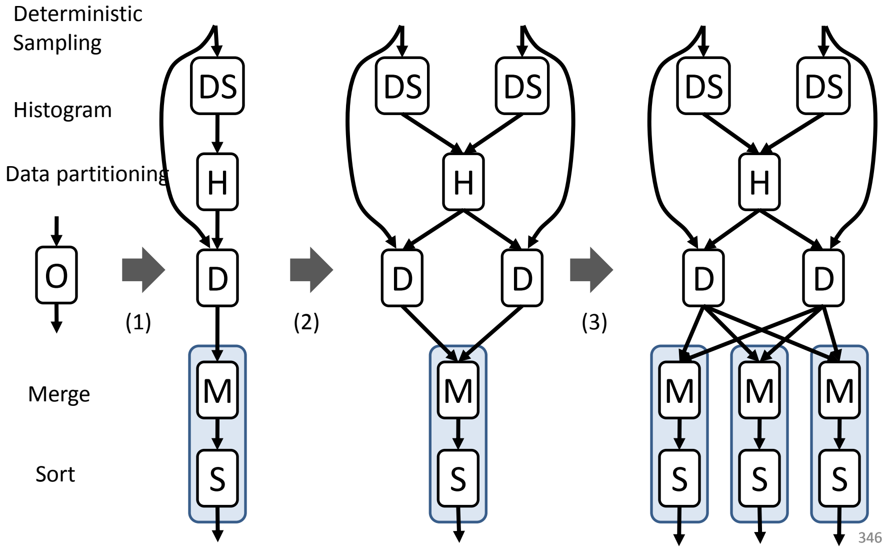


344

# Distributed Sorting in DryadLINQ

```
public static IQueryable<TSource>
DSort<TSource, TKey>(this IQueryable<TSource> source,
                        Expression<Func<TSource, TKey>> keySelector,
                        int pcount)
{
        var samples = source.Apply(x => Sampling(x));
        var keys = samples.Apply(x => ComputeKeys(x, pcount));
        var parts = source.RangePartition(keySelector, keys);
        return parts.OrderBy(keySelector);
}
```

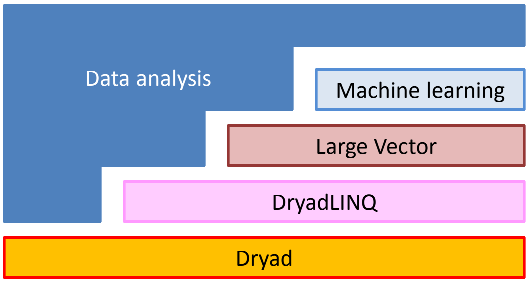
# Distributed Sorting Plan



# Outline

- Introduction
- Dryad
- DryadLINQ
- Building on DryadLINQ

# Machine Learning in DryadLINQ



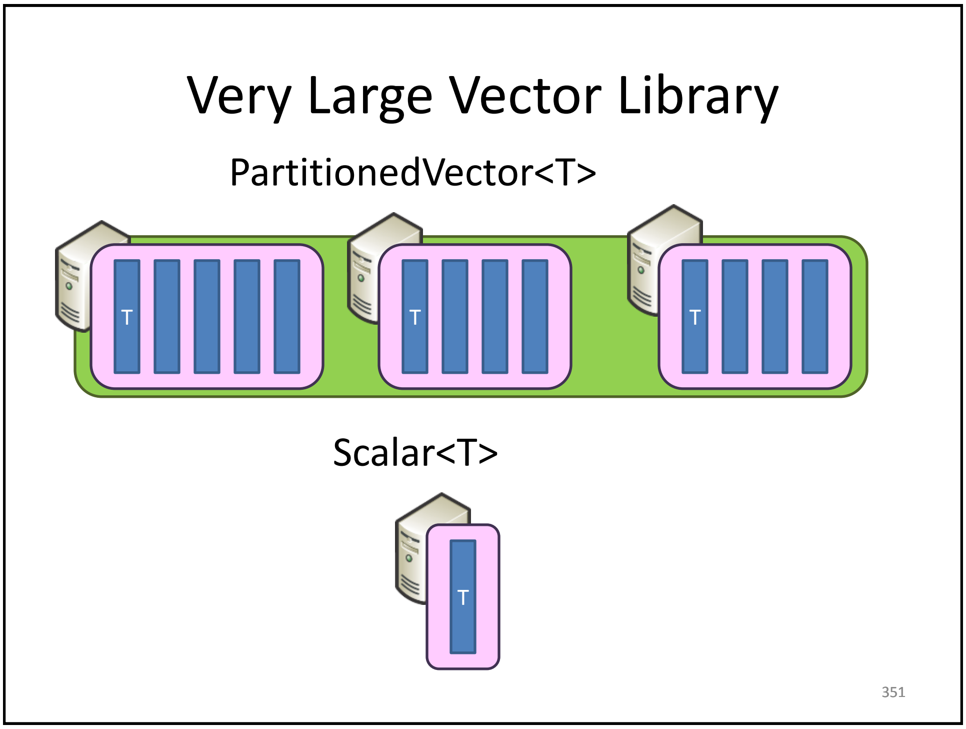
Data analysis

Machine learning

Large Vector

DryadLINQ

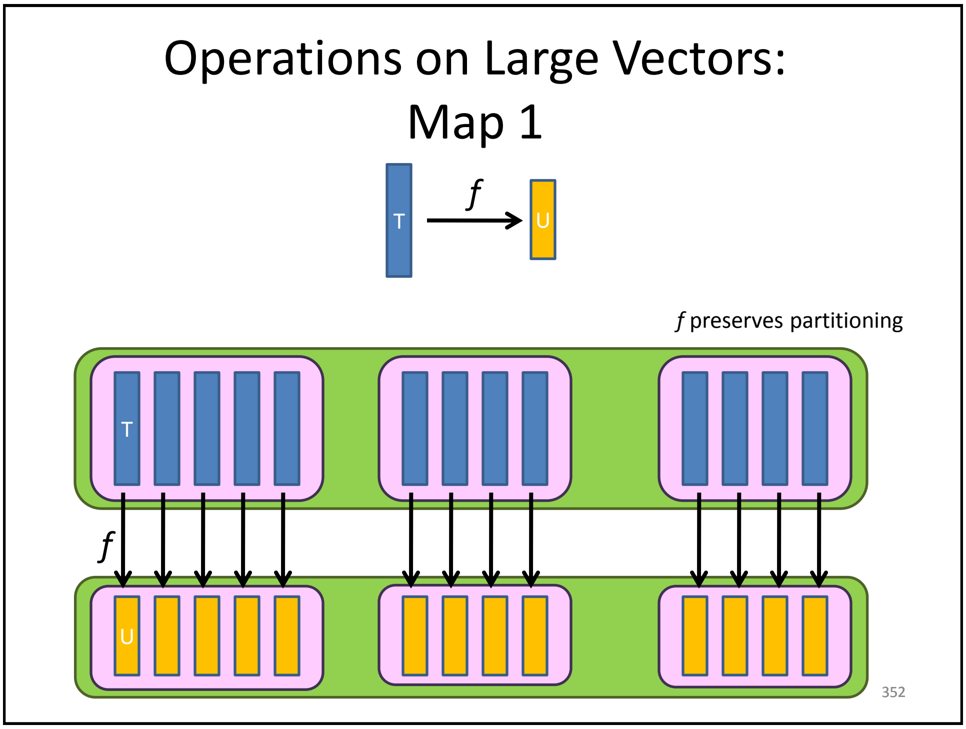
Dryad



# Very Large Vector Library

PartitionedVector<T>

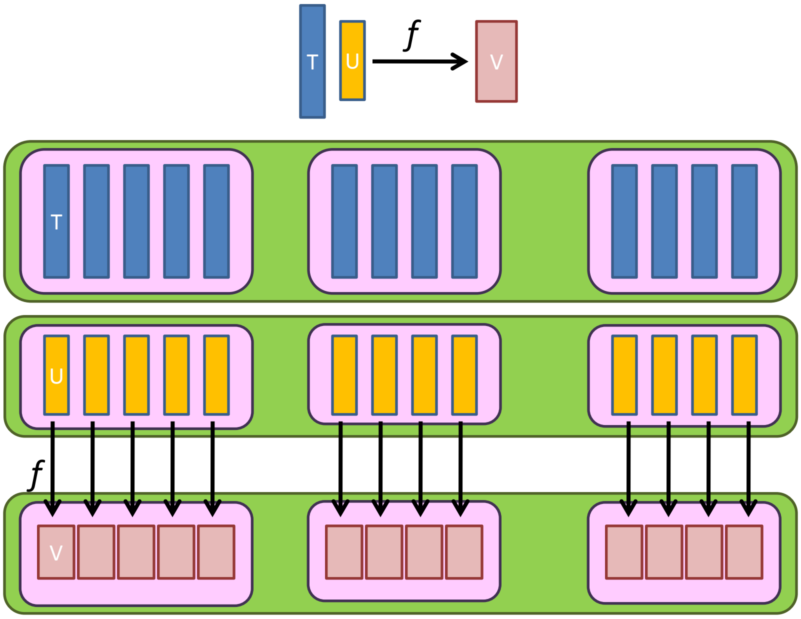
Scalar<T>



# Operations on Large Vectors: Map 1

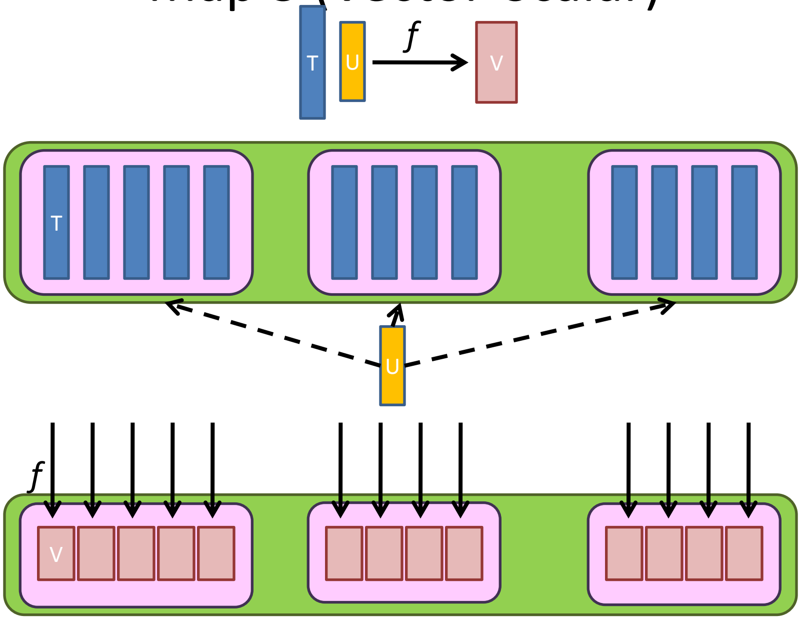
*f* preserves partitioning

## Map 2 (Pairwise)



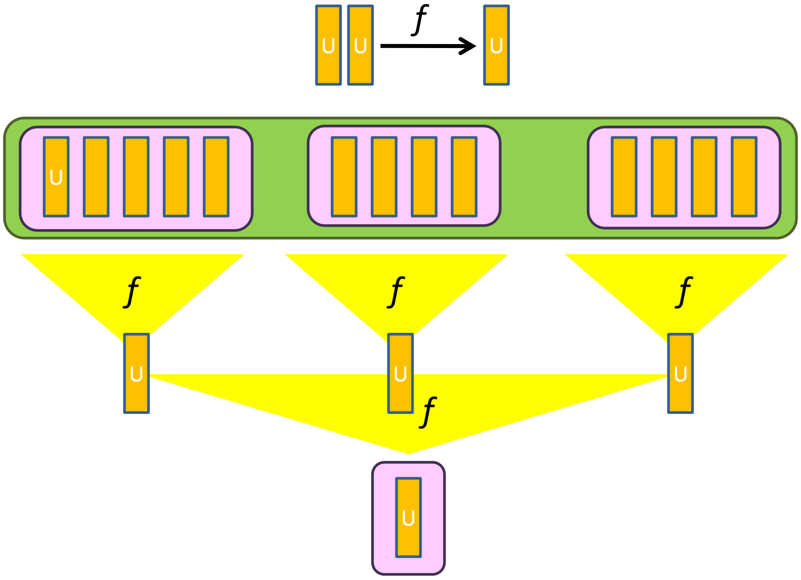
353

## Map 3 (Vector-Scalar)

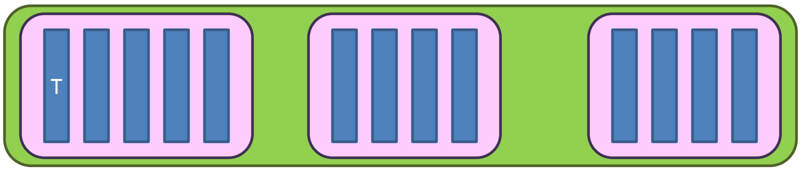


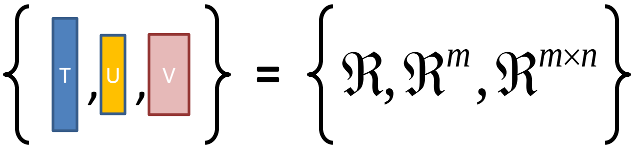
354

# Reduce (Fold)



# Linear Algebra





# Linear Regression

- Data

$$x_t \in \Re^n, y_t \in \Re^m \qquad t \in \{1,\ldots,n\}$$

- Find

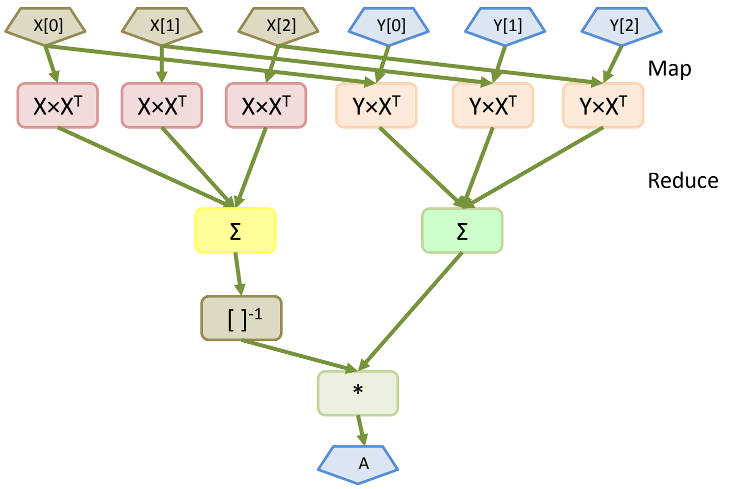
$$A \in \Re^{n \times m}$$

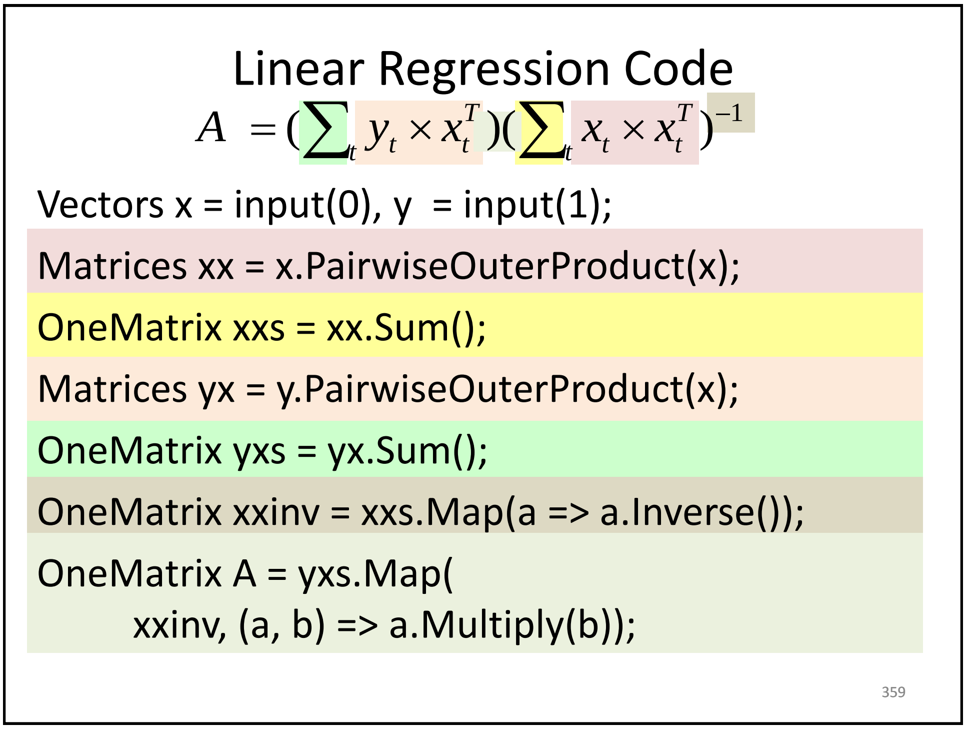
- S.t.

$$Ax_t \approx y_t$$

# Analytic Solution

$$A = (\sum_t y_t \times x_t^T)(\sum_t x_t \times x_t^T)^{-1}$$



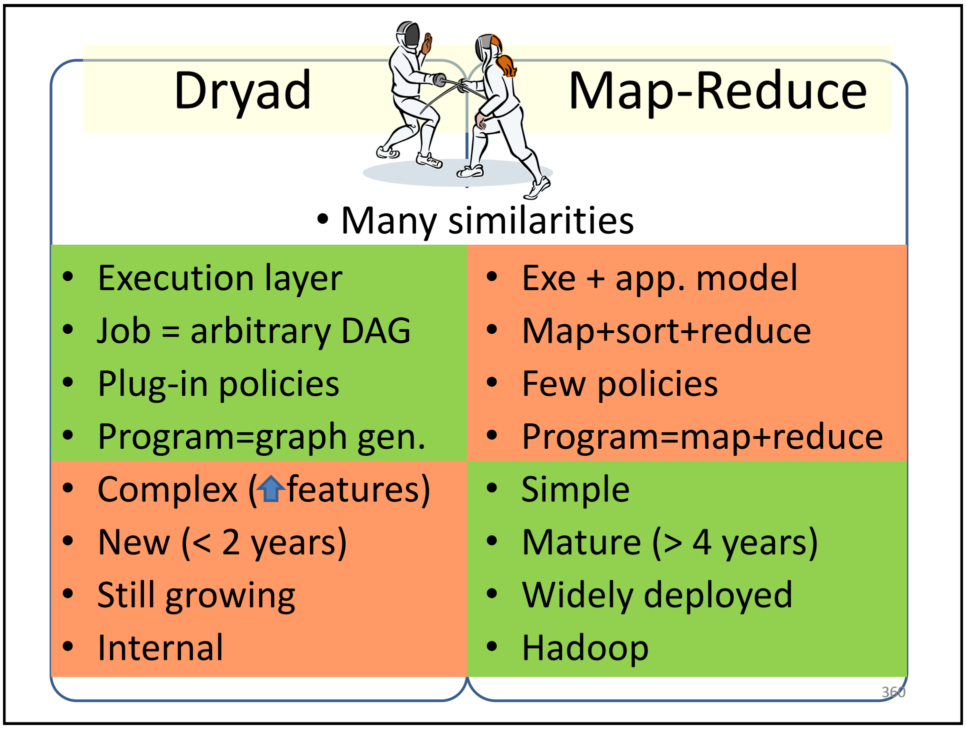


# Linear Regression Code

$$A = (\sum_t y_t \times x_t^T)(\sum_t x_t \times x_t^T)^{-1}$$

```
Vectors x = input(0), y  = input(1);
Matrices xx = x.PairwiseOuterProduct(x);
OneMatrix xxs = xx.Sum();
Matrices yx = y.PairwiseOuterProduct(x);
OneMatrix yxs = yx.Sum();
OneMatrix xxinv = xxs.Map(a => a.Inverse());
OneMatrix A = yxs.Map(
        xxinv, (a, b) => a.Multiply(b));
```

359

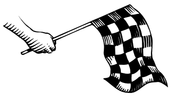


- Many similarities

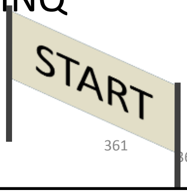
| Dryad | Map-Reduce |
|---|---|
| • Execution layer | • Exe + app. model |
| • Job = arbitrary DAG | • Map+sort+reduce |
| • Plug-in policies | • Few policies |
| • Program=graph gen. | • Program=map+reduce |
| • Complex (↑features) | • Simple |
| • New (< 2 years) | • Mature (> 4 years) |
| • Still growing | • Widely deployed |
| • Internal | • Hadoop |

360

# Conclusions



- Dryad = distributed execution environment
- Application-independent (semantics oblivious)
- Supports rich software ecosystem
  – Relational algebra
  – Map-reduce
  – LINQ
  – Etc.
- DryadLINQ = A Dryad provider for LINQ
- This is only the beginning!



361

Finally, let us put things into perspective by looking at alternatives to MapReduce.

We started with Dryad from Microsoft, now move on to parallel and distributed databases.

# Parallel Database Systems

- Data: relations
- Relational operators process relations and output relations
  - Selection
  - Projection
  - Join
  - Group By and aggregation
- Query language: SQL

# SQL

- Declarative language
  - Specify what you want, not how to get it
- Database optimizer chooses best implementation
  - Query plan: DAG of operators and their implementations
  - Minimize cost of query plan
    - I/O cost, CPU cost
  - Optimizer explores space of query plans, chooses best one

# SQL in Parallel

- Same query, just replace optimizer
  - Take data location and network cost into account
  - Optimize for latency or total cost
- Add new operators
  - Exchange operator: behaves like an iterator, but receives input via inter-process communication rather than iterator procedure calls
  - Split and Merge: create and join parallel dataflows
- Add new operator implementations
  - Semi-join implementation to reduce network communication cost
- The optimizer is more complex, but SQL does not need to change

365

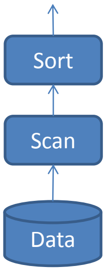
# Distributed Query Optimization

- Start: calculus query on global relations
- Transform into algebraic query on global relations
- Perform data localization, using fragment schema, to generate algebraic query on fragments
- Perform global optimization to create distributed query execution plan
- Run on local sites in parallel

366

## Pipeline Parallelism

- Computation of one operator proceeds in parallel with another
- Model: output pulls from last operators, which pulls from its inputs and so on

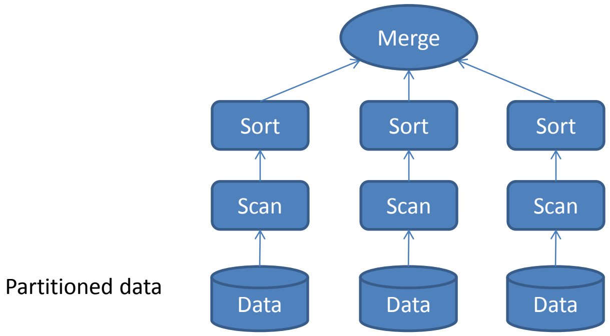


## Limited Benefits of Pipeline Parallelism

- Relational pipelines are usually not very long
  - Ten or longer is rare
- Some operators are blocking and cannot be pipelined
  - Aggregates, sorting
- Execution cost of one operator might be much larger than the others
  - Limits speedup obtained by pipelining

# Partitioned Parallelism

- Query performs batch-style computation on many input tuples



# Data Partitioning

- Round-robin
  - Simple, but not helpful for associative access
- Hash partitioning
  - Assign tuples to partition using hash function
  - Good for associative access (equality-based)
  - Not good for range queries
- Range partitioning
  - Partition data into continuous ranges
  - Good for range queries, parallel sort
  - Risks data skew (uneven partitions) and execution skew (uneven access pattern)

# Distributed Transactions?

- Transactions were crucial for the success of database systems
- Enable concurrent processing of multiple queries, but programmers could write them as if they executed in isolation

# The ACID Properties

- Atomicity: Either all or none of the transaction's actions are executed
  - Even when a crash occurs mid-way
- Consistency: Transaction run by itself must preserve consistency of the database
  - User's responsibility
- Isolation: Transaction semantics do not depend on other concurrently executed transactions
- Durability: Effects of successfully committed transactions should persist, even when crashes occur

# Example

| | | | | |
|---|---|---|---|---|
| T1: | BEGIN | A=A+100, | B=B-100 | END |
| T2: | BEGIN | A=1.06*A, | B=1.06*B | END |

- T1 transfers $100 from B's account to A's account.
- T2 credits both accounts with a 6% interest payment.
- There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together.
- However, the net effect must be equivalent to these two transactions running serially in some order.

# Example (Contd.)

- Consider a possible interleaving (schedule):

| | | | | |
|---|---|---|---|---|
| T1: | A=A+100, | | B=B-100 | |
| T2: | | A=1.06*A, | | B=1.06*B |

- This is OK. But what about:

| | | | |
|---|---|---|---|
| T1: | A=A+100, | | B=B-100 |
| T2: | | A=1.06*A, B=1.06*B | |

- The DBMS's view of the second schedule:

| | | | |
|---|---|---|---|
| T1: | R(A), W(A), | | R(B), W(B) |
| T2: | | R(A), W(A), R(B), W(B) | |

## Scheduling Transactions

- Serial schedule: Schedule that does not interleave the actions of different transactions.
  - Easy for programmer, easy to achieve consistency
  - Bad for performance
- Equivalent schedules: For any database state, the effect (on the objects in the database) of executing the first schedule is identical to the effect of executing the second schedule.
- Serializable schedule: A schedule that is equivalent to some serial execution of the transactions.
  - Retains advantages of serial schedule, but addresses performance issue

## Anomalies with Interleaved Execution

| | | | |
|---|---|---|---|
| T1: | R(A), W(A), | | R(B), W(B), Abort |
| T2: | | R(A), W(A), C | |

- Reading Uncommitted Data (WR Conflicts, “dirty reads”)
- Example: T1(A=A-100), T2(A=1.06A), T2(B=1.06B), C(T2), T1(B=B+100)
- T2 reads value A written by T1 before T1 completed its changes
- If T1 later aborts, T2 worked with invalid data

# More Anomalies

```
T1:     R(A),                        R(A), W(A), C
T2:              R(A), W(A), C
```

- Unrepeatable Reads (RW Conflicts)
- T1 sees two different values of A, even though it did not change A between the reads
- Example: online bookstore
  - Only one copy of a book left
  - Both T1 and T2 see that 1 copy is left, then try to order
  - T1 gets an error message when trying to order
  - Could not have happened with serial execution

# Even More Anomalies

```
T1:     W(A),                     W(B), C
T2:              W(A), W(B), C
```

- Overwriting Uncommitted Data (WW Conflicts)
- T1's B and T2's A persist, which would not happen with any serial execution
- Example: 2 people with same salary
  - T1 sets both salaries to 2000, T2 sets both to 1000
  - Above schedule results in A=1000, B=2000, which is inconsistent

## Aborted Transactions

- All actions of aborted transactions have to be undone
- Dirty read can result in unrecoverable schedule
  - T1 writes A, then T2 reads A and makes modifications based on A's value
  - T2 commits, and later T1 is aborted
  - T2 worked with invalid data and hence has to be aborted as well; but T2 already committed…
- Recoverable schedule: cannot allow T2 to commit until T1 has committed
  - Can lead to cascading aborts

## Preventing Anomalies through Locking

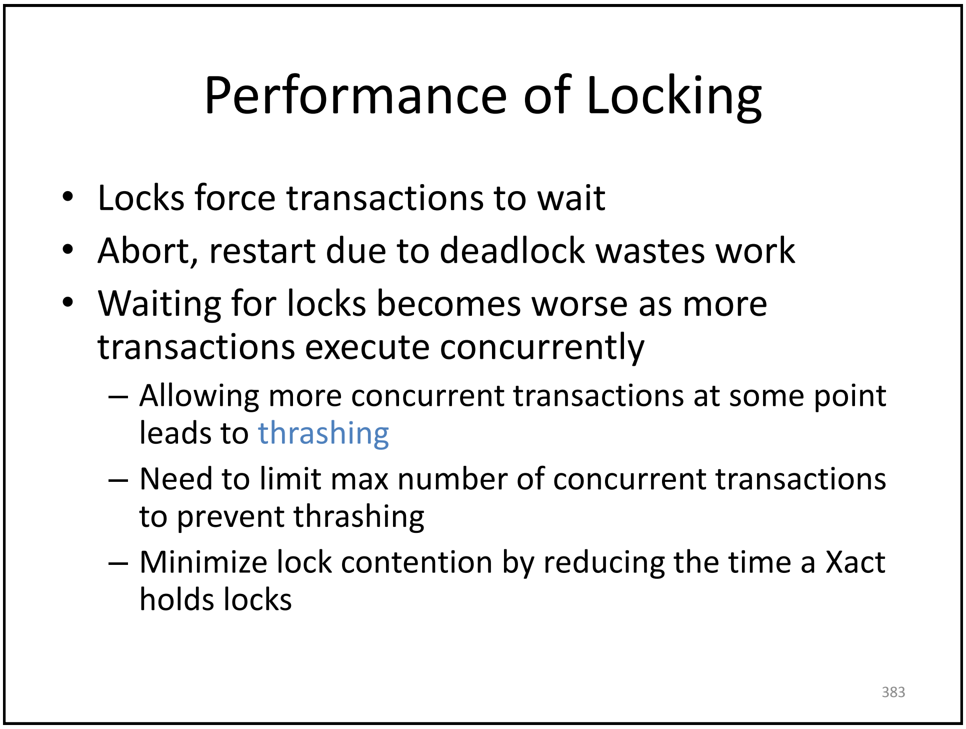
- DBMS can support concurrent transactions while preventing anomalies by using a locking protocol
- If a transaction wants to read an object, it first requests a shared lock (S-lock) on the object
- If a transaction wants to modify an object, it first requests an exclusive lock (X-lock) on the object
- Multiple transactions can hold a shared lock on an object
- At most one transaction can hold an exclusive lock on an object

## Lock-Based Concurrency Control

- Strict Two-phase Locking (Strict 2PL) Protocol:
  - Each Xact must obtain the appropriate lock before accessing an object.
  - All locks held by a transaction are released when the transaction is completed.
  - All this happens automatically inside the DBMS
- Strict 2PL allows only serializable schedules.
  - Prevents all the anomalies shown earlier

## Deadlocks

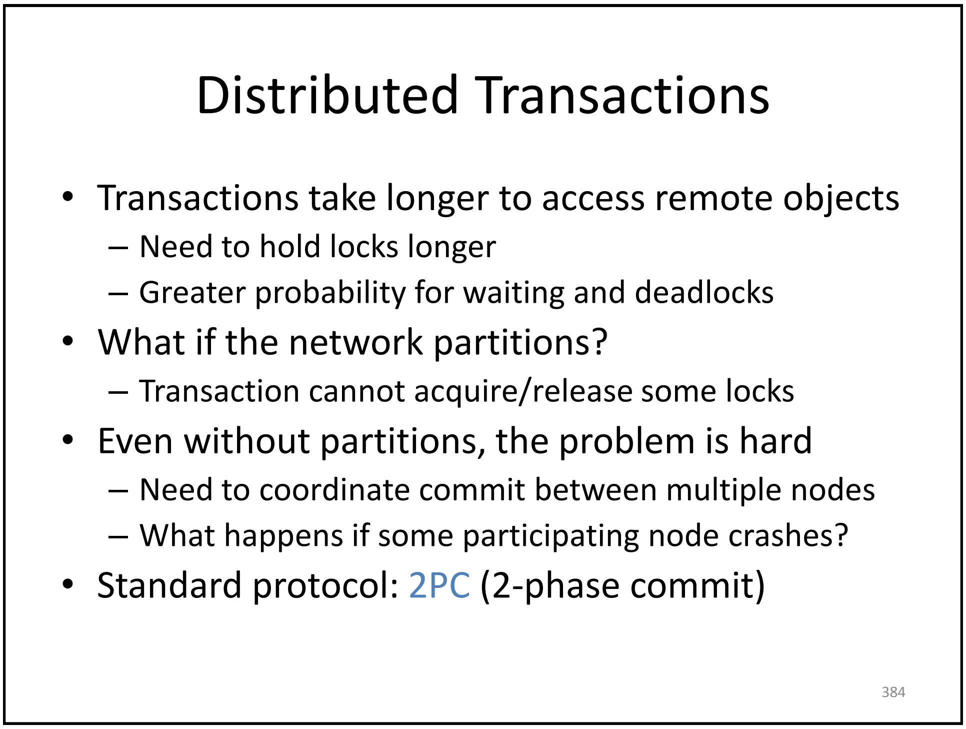
- Assume T1 and T2 both want to read and write objects A and B
  - T1 acquires X-lock on A; T2 acquires X-lock on B
  - Now T1 wants to update B, but has to wait for T2 to release its lock on B
  - But T2 wants to read A and also waits for T1 to release its lock on A
  - Strict 2PL does not allow either to release its locks before the transaction completed. Deadlock!
- DBMS can detect this
  - Automatically breaks deadlock by aborting one of the involved transactions



# Performance of Locking

- Locks force transactions to wait
- Abort, restart due to deadlock wastes work
- Waiting for locks becomes worse as more transactions execute concurrently
  - Allowing more concurrent transactions at some point leads to thrashing
  - Need to limit max number of concurrent transactions to prevent thrashing
  - Minimize lock contention by reducing the time a Xact holds locks

383



# Distributed Transactions

- Transactions take longer to access remote objects
  - Need to hold locks longer
  - Greater probability for waiting and deadlocks
- What if the network partitions?
  - Transaction cannot acquire/release some locks
- Even without partitions, the problem is hard
  - Need to coordinate commit between multiple nodes
  - What happens if some participating node crashes?
- Standard protocol: 2PC (2-phase commit)

384

# 2PC Basics

- Commit-request phase
  - Coordinator asks all participants to prepare for commit
  - Participants vote YES or NO to commit request
- Commit phase
  - Based on participants' votes, coordinator decides to commit (if all voted YES) or abort
  - Coordinator notifies participants about decision
  - Participants apply corresponding action (commit or abort) locally

# 2PC Problems

- 2PC = blocking protocol
  - Nodes cannot make a decision without hearing from coordinator, e.g., might hold on to locks forever if coordinator is down and they answered YES to first request
- Expensive for many-worker transactions
- Some issues were addressed by later 2PC modifications, but the basic problems remain