Algorithmic Infrastructure and Performance of Finding Optimal Conditional Sparse l_p-norm Regression

John Hainline

Outline

- Problem Overview
- Formulation
- Algorithms
- Implementation
- Architecture
- Evaluation
- Conclusion

Problem Overview

- Goals
  - Provide scalable and robust implementation of new algorithm in machine learning
  - Specifically the "Conditional Sparse l_p-norm Regression Problem" (explained next)
  - Focused on engineering a performant parallelized cloud solution
  - Created database to provide complete information recovery

Problem Overview

We are trying to implement an algorithm that solves a Conditional Sparse l_p-norm Regression problem

- Given $X \in \mathbb{B}^{n \times p} = (\vec{x}_1, \vec{x}_1, \ldots, \vec{x}_n)^T$
  $Y \in \mathbb{R}^{n \times m} = (\vec{y}_1, \vec{y}_2, \ldots, \vec{y}_n)^T$
  $Z \in \mathbb{R}^n$
- Find conditional **c** (our k-DNF) such that **c(X)** creates a subpopulation where the regression line $\langle \vec{a}, Y \rangle$ predicts **Z**
  - **a** is sparse, i.e. uses few components of **Y** (e.g. 2)
- Using l2-norm (i.e. least squared error)

Problem Overview

- k Disjunctive Normal Form (k-DNF)
    - A formula is a DNF iff it is a disjunction (∨) of conjunctions (∧) of literals
    - A **k**-DNF means all conjunctions contain **k** literals
    - Valid 2-DNFs
        - $(x1 \land x2)$
        - $(x8 \land \neg x2) \lor (\neg x2 \land x5) \lor (\neg x1 \land x2) \lor (x3 \land x4)$
    - Valid 3-DNFs
        - $(x1 \land x2 \land x3)$
        - $(x1 \land \neg x2 \land \neg x3) \lor (\neg x1 \land \neg x2 \land x5) \lor (\neg x2 \land x3 \land x4)$
    - Using 2-DNFs although system could easily be extended for arbitrary k-DNFs

"Red-Blue Set Cover" Formulation

- Working with real-valued variables, and using a greedy strategy
  - "Sets" are all possible terms **(xi ∧ xj)** of which there are: $4 \times \binom{p}{2}$
  - i.e. each term denotes a set of some points of **Y**
  - Originally couched in terms of red or blue categories for points
    - Required to cover some number of blue points
    - Number of red points covered is the "cost"
  - Construct **c(X)** (our 2-DNF) such that we:
    - Minimize "redness" of points (i.e. distance from regression line $\langle \vec{a}, Y \rangle$)
    - Maximize number of points

Optimizations

- Sub-sampling
  - Reduce size of **Y** using sub-sampling (**n**→**r**)
  - High probability of finding hidden k-DNFs
- Combinatorics
  - Must compute Set Cover $\binom{r}{2} \times \binom{m}{2}$ times

Algorithms

Algorithm 1: This algorithm breaks down the relevant search space of $\boldsymbol{X} \in \mathbb{B}^{n \times p}$ and $\boldsymbol{Y} \in \mathbb{R}^{n \times m}$, giving SetCover the parameters it needs. Note that $\boldsymbol{terms}$ must include the $\neg$ versions of our $\binom{p}{2}$ possible combinations of $\boldsymbol{X}$, which is why we have $4 \cdot \binom{p}{2}$ combinations.

```
def ComputeBestKDNF(X, Y, Z, r):
    X, Y = SubsamplePoints(X, Y, r)
    terms = (x_i ∩ x_j) for all 4 · (p choose 2) combinations of X
    best_terms = ∅
    best_error = ∞
    for (i, j) ∈ (r choose 2) points do
        for (k, l) ∈ (m choose 2) dimensions do
            t, e = SetCover(X, Y, Z, terms, i, j, k, l)
            if e < best_error then
                best_terms = t
                best_error = e
            end
        end
    end
    return best_terms, best_error
```

Algorithms

Algorithm 2: The SetCover algorithm searches for the best k-DNF (lowest error in terms of "redness") for a given pair of data points i, j and dimensions k, l in $Y \in \mathbb{R}^{n \times m}$. The variable $terms$ holds every *set* generated by our $\binom{p}{2}$ terms $(x_i \cap x_j)$ in $X \in \mathbb{B}^{n \times p}$. μ dictates how large of a sub-population we are looking for.

def *SetCover*(X, Y, Z, $terms$, i, j, k, l)**:**

- $\vec{a} = \text{CalculateRegressionLine}(\vec{y}[k]_i, \vec{y}[l]_j)$
- $r\_points = \text{CalculateRednessForPoints}(Y, \vec{a})$
- $r\_terms = \text{CalculateRednessForTerms}(terms, r\_points)$
- $sort = \text{SortByAverageRedness}(r\_terms)$
- $best\_terms = \emptyset$
- $best\_error = \infty$
- $cur\_terms = \emptyset$
- **while** $(cur\_terms.size < \mu \cdot n)$ **do**
    - $next = \text{NextUnusedTermInSort}(sort, cur\_terms)$
    - $cur\_terms.add(next)$
- **end**
- $cur\_error = \text{AverageRednessPerPoint}(cur\_terms)$
- **if** $cur\_error < best\_error$ **then**
    - $best\_error = cur\_error$
    - $best\_terms = cur\_terms$
- **end**
- **return** $best\_terms, best\_error$

Implementation

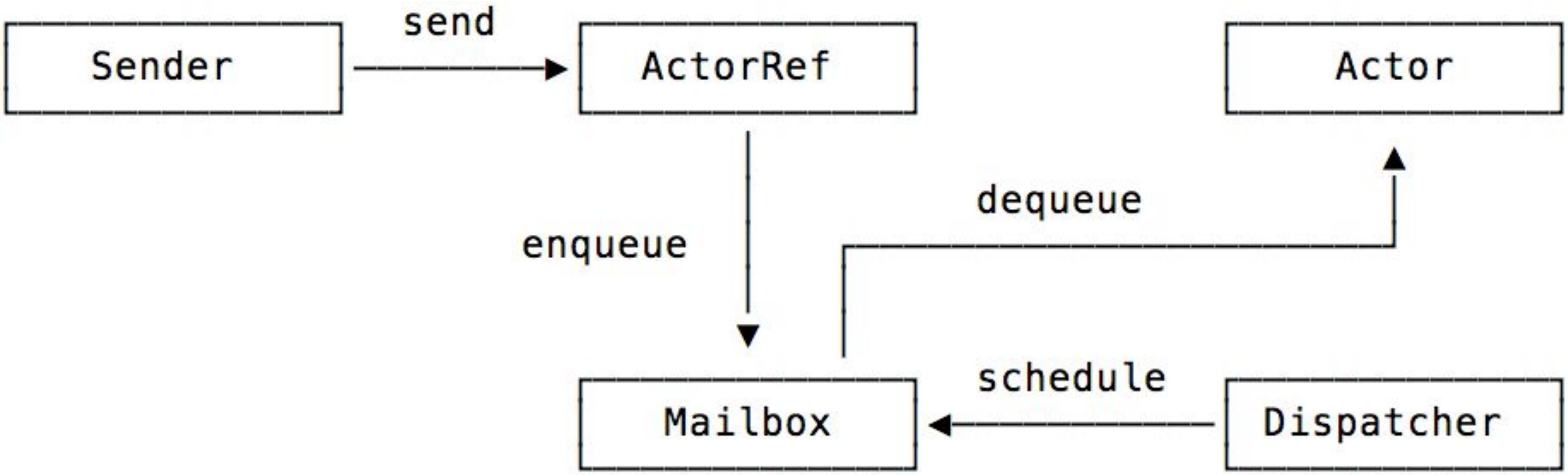
- Akka Library
  - Akka Actors
  - Akka Streams
  - Akka Http
  - Akka Cluster
  - Cluster Sharding
  - Distributed Data
  - Akka Persistence
  - Alpakka
  - Akka gRPC
  - Akka Management

Implementation

- Akka Library
  - **Akka Actors**
  - **Akka Streams**
  - Akka Http
  - **Akka Cluster**
  - Cluster Sharding
  - Distributed Data
  - Akka Persistence
  - Alpakka
  - Akka gRPC
  - Akka Management

Implementation

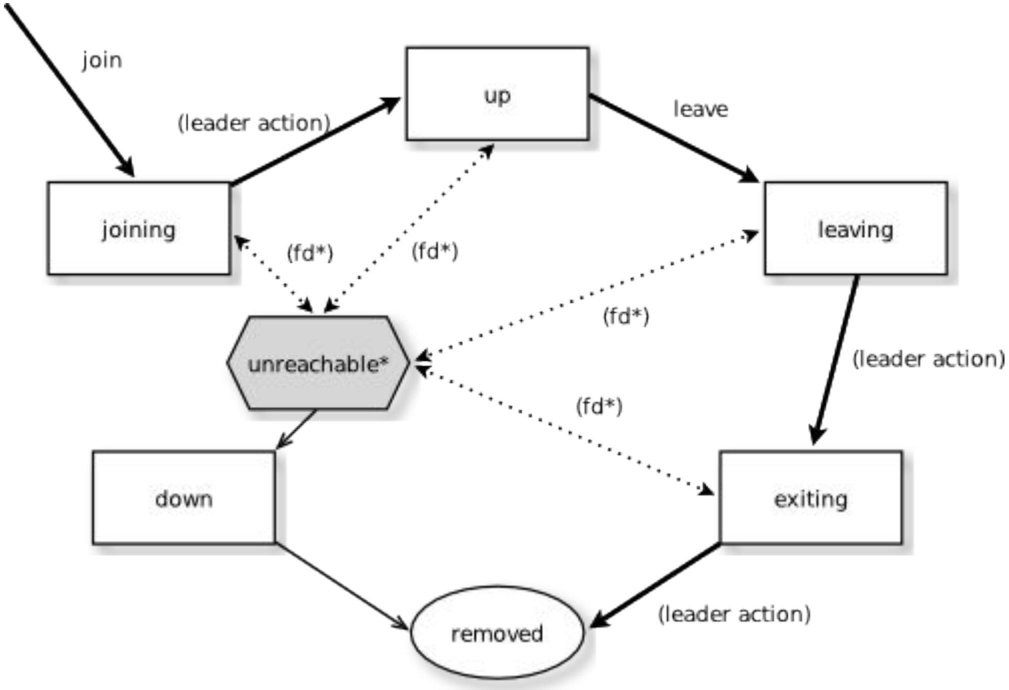
- Akka Actors



Source: https://blog.codecentric.de/en/2015/08/introduction-to-akka-actors/

Implementation

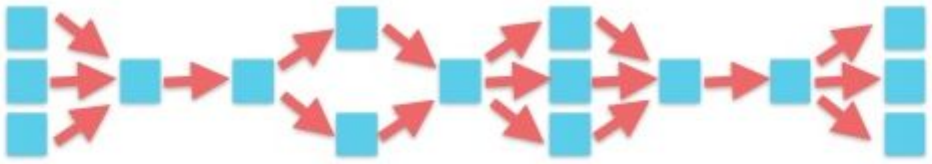
- Akka Cluster
  - Member States
    - Joining
    - Up
    - Leaving
    - Exiting
    - Removed
    - Down

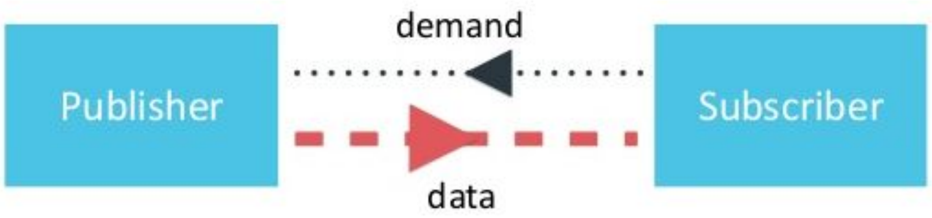


Source: https://doc.akka.io/docs/akka/2.5/common/cluster.html

Implementation

- Akka Streams
  - Graph describes movement of data
  - Starts at *Source(s)*
  - Travels through arbitrarily complex graph of *Flows*
  - Ends at *Sink(s)*
- Back-Pressure
  - *Sink* demands data
  - Demand moves backward to *Source*
  - *Source* pushes data
  - This manages stream speed and keeps nodes from desynchronizing

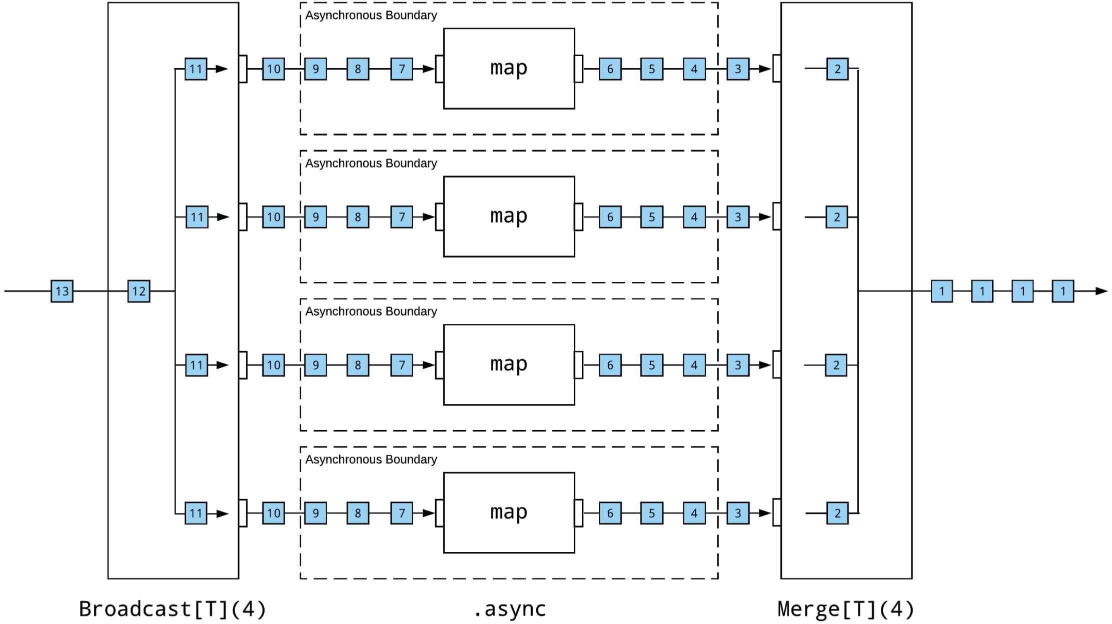




Source for images:
https://www.slideshare.net/Typesafe_Inc/reactive-streams-100-and-why-you-should-care-webinar

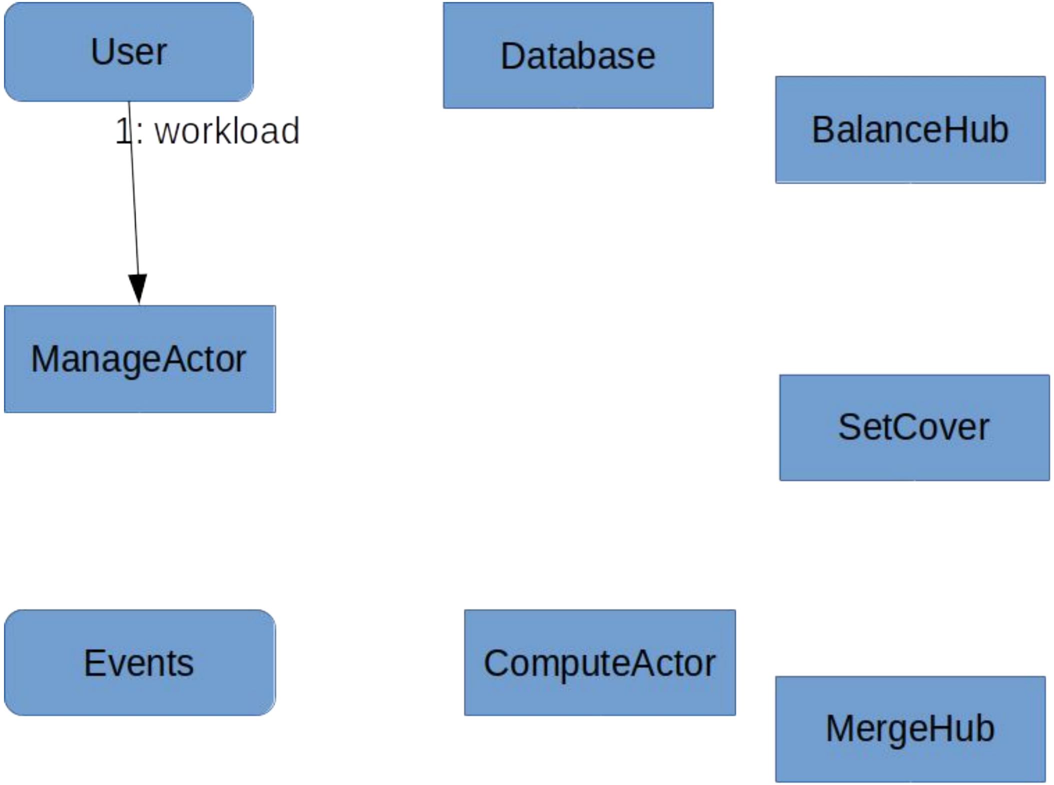
Implementation

- Akka Streams
  - *Source* sends **work** to *BalanceHub*
  - *BalanceHub* broadcasts **work** to *SetCover* instances
  - *SetCover* runs algorithm, sending on **results**
  - *MergeHub* merges **results** to a single stream and saves

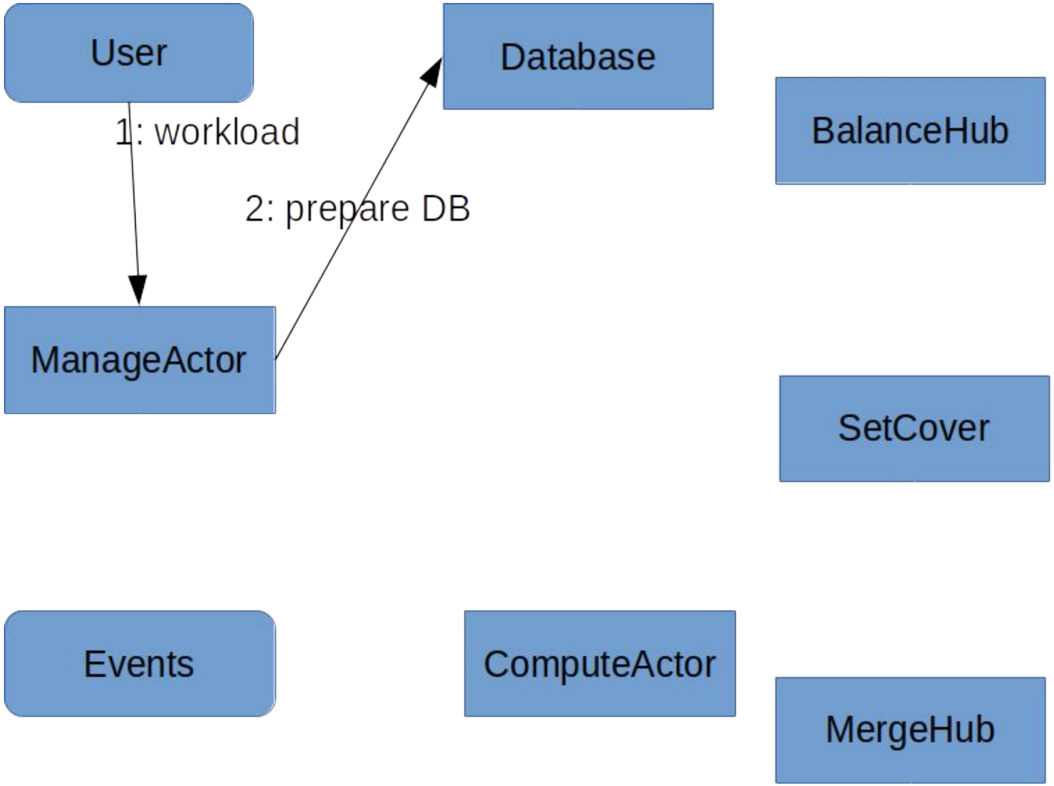


Source: https://blog.colinbreck.com/partitioning-akka-streams-to-maximize-throughput/

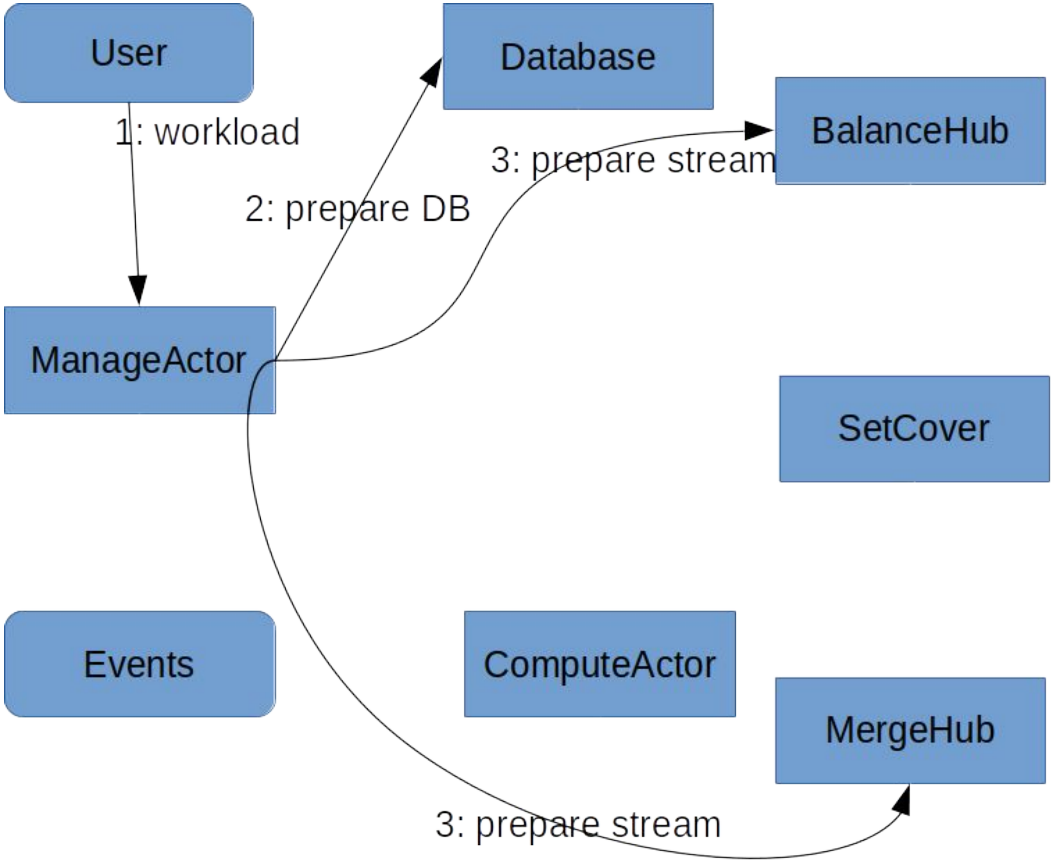
Architecture



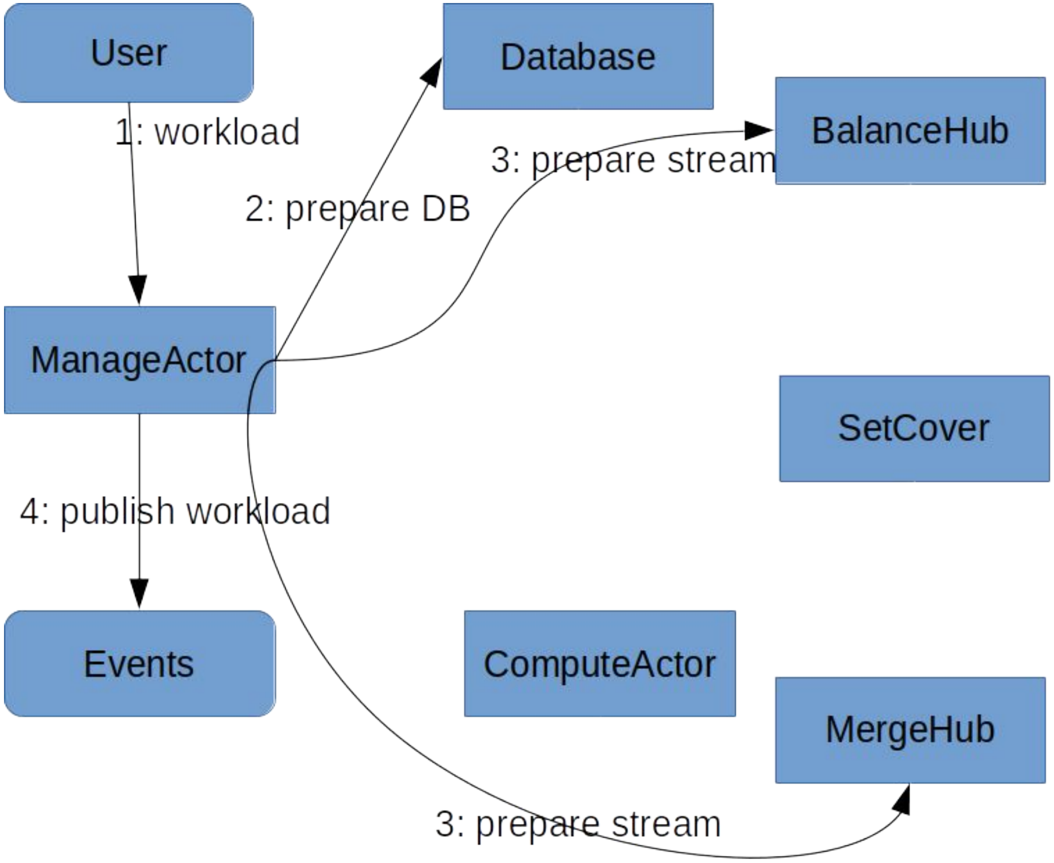
Architecture



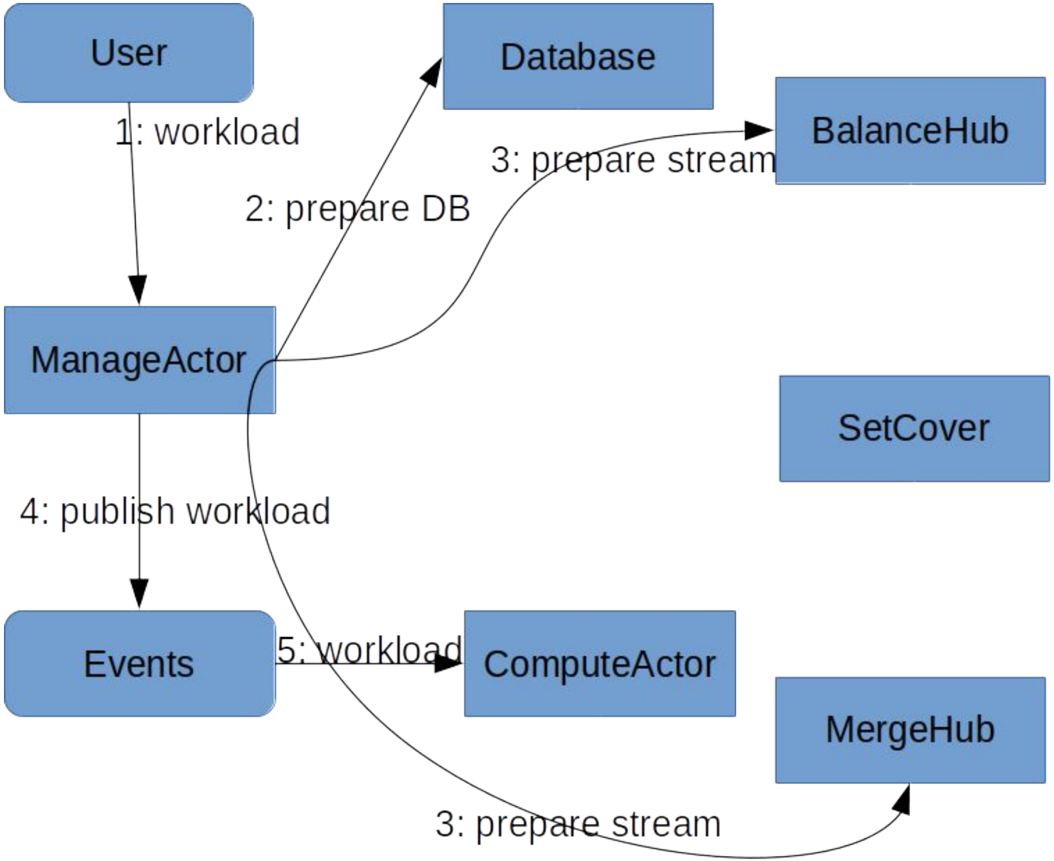
Architecture



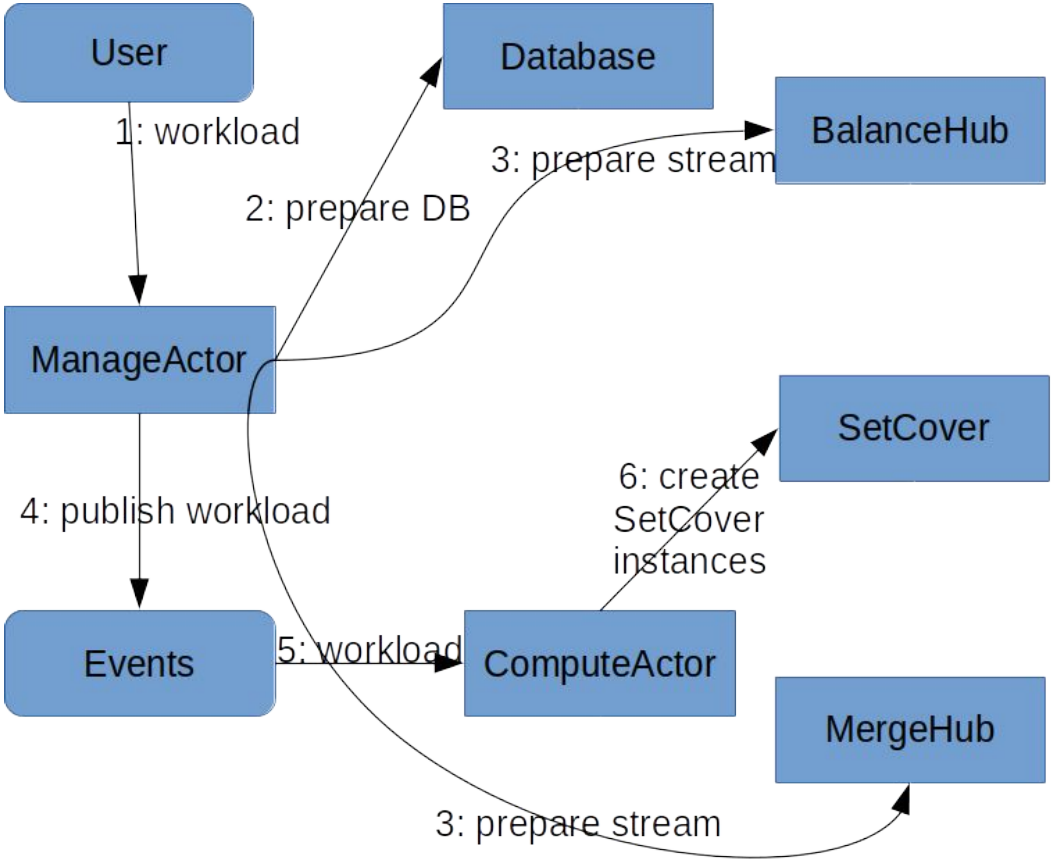
Architecture



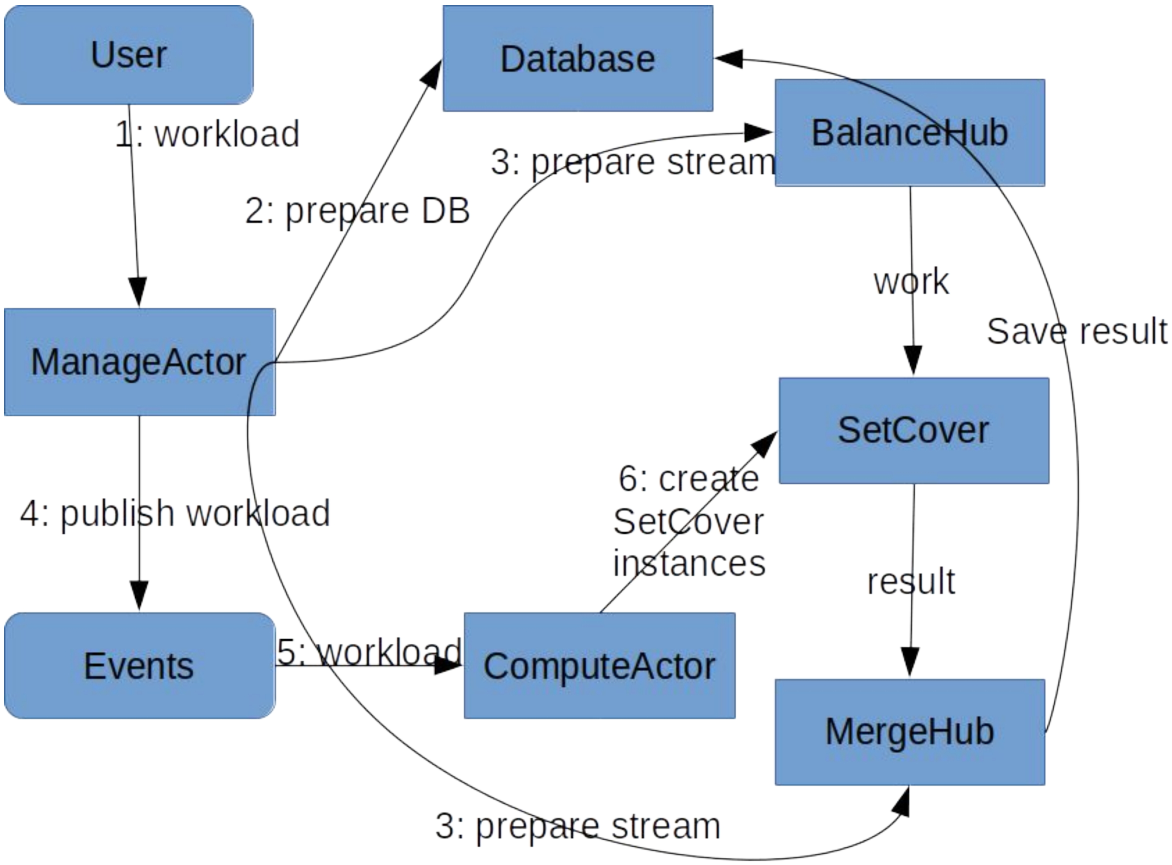
Architecture



Architecture



Architecture



Architecture

- Workload
  - name: data .sql filename
  - mu: % of data which k-DNF includes
  - optionalSubset: number of points to select from **Y**
  - optionalRandomSeed: RNG seed for selecting **Y** points
- Work
  - index: absolute "position" in Co-Lex order
  - selectedDimensions: (k, l)
  - selectedRows: (i, j)
- Result
  - index: absolute "position"
  - selectedDimensions: (k, l)
  - selectedRows: (i, j)
  - coefficients: sparse regression line
  - kDNF: (x1 ∩ ¬x2) ∪ (x3 ∩ x5)

```
case class Workload(name: String, mu: Double,
                    optionalSubset: Option[Int] = None,
                    optionalRandomSeed: Option[Int] = None)

case class Work(index: Int,
                selectedDimensions: Array[Int],
                selectedRows: Array[Int])

case class Result(index: Int,
                  dimensions: Array[Int],
                  rows: Array[Int],
                  coefficients: Array[Double],
                  error: Option[Double],
                  kDNF: Option[String])
```

Evaluation

- Sub-sampling increased speed by multiple orders of magnitude
  - Uses a RNG seed so we can recover a failed workload
- BitSet implementation gave 10x speed increase
  - Went from using 30 CPUs to 100 CPUs
- Batching (in streams) gave 10x speed increase
  - Went from using 100 CPUs to 200+ CPUs
- Expectation
  - Likely scales to 300+ CPUs
  - Not tested due to cost on GKE

Conclusion

- System dynamically links SetCover Flows into Akka Stream
  - Allows scaling across an entire cluster
- Using Sub-sampling, Batching, and BitSet for speed
- MySQL Database allows recovery on system failure
- Configured to run on Google Kubernetes Engine (or a local machine)
- Successfully ran
  - **n** = 10,000 points, sub-sampled to **s** = 500
  - SetCover ran 27,702,500 times over 5 days
  - Used 208 CPUs across 26 VMs

Conclusion

Questions?