A Configurable Application View Storage System

Sibel Adali

Rensselaer Polytechnic Institute

Overview of talk

- Overview
- Related work
- Motivation –information integration
- Storage management
  - View re-use
  - View eviction
- Experimental results, preliminary findings
- An example view management problem
- Research problems

CAVES

- General purpose storage management system for networked applications which includes a “storage shell” that incorporates
  - Programmable storage management protocols
  - Complex view re-use options
  - Dynamic tuning of protocols to actual system loads
  - Verification modules for the effectiveness of the overall system

CAVES –challenges

- Unit of storage is not uniform → views of varying sizes
- Storage system may need to serve multiple types of queries with different characteristics → shared storage pool
- Characteristics of actual requests are unpredictable → unknown storage policies
- The cost parameters may change from application to application → different optimization problems

Storage Management

- Client-side caching of views –instead of tuple based caching [Franklin et. al.]
- Semantic caching [Franklin, Chu, Godfrey, …]
- OLAP/ Data Cube applications
  - WATCHMAN [Scheuermann]
  - DYNA-MAT [Kotidis]
  - Proxy-server architectures for OLAP [Kalnis, Papadias]

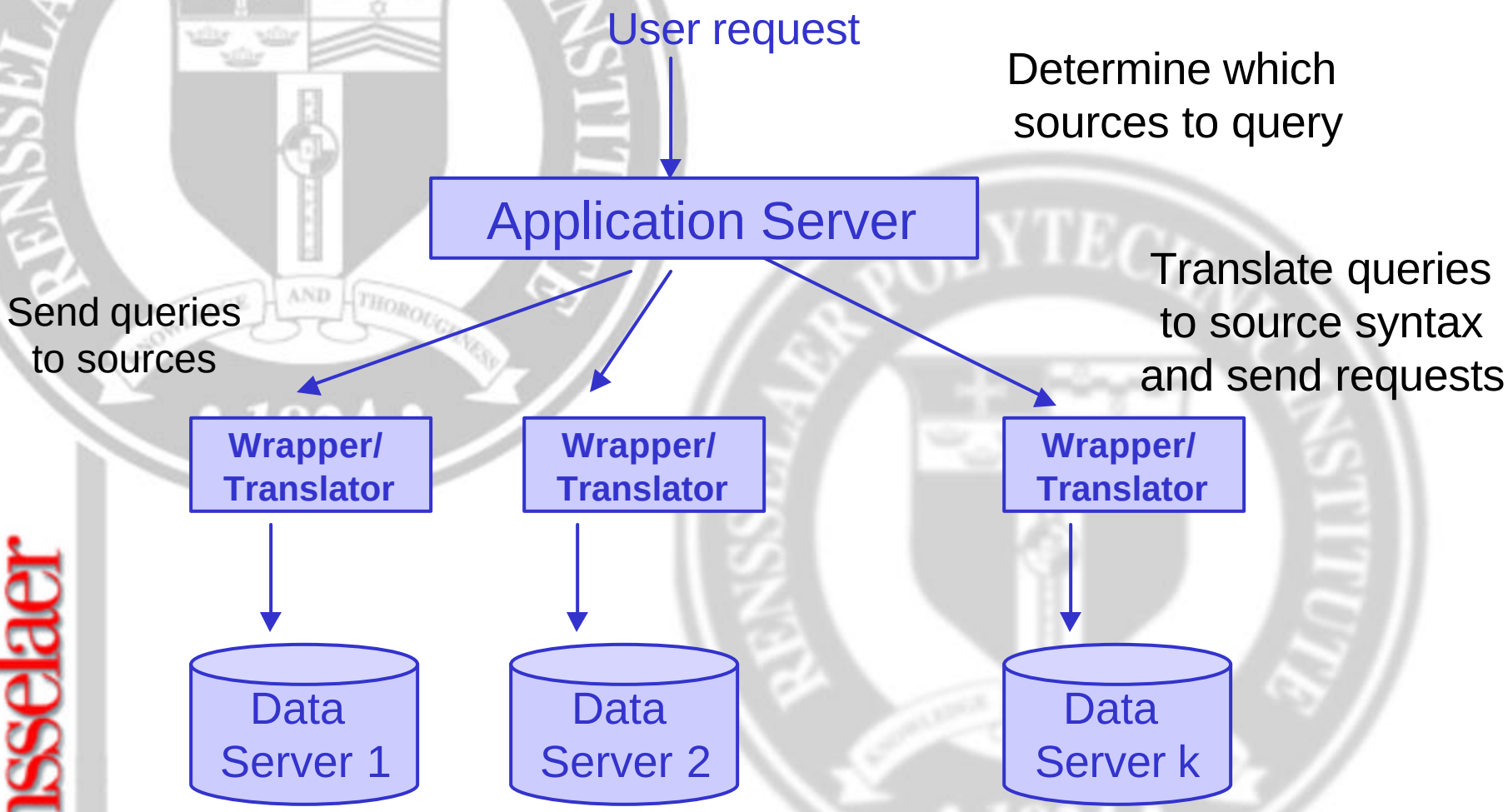
CAVES - challenges

1. Are any common tasks a storage management system has to perform?
2. How much does an application programmer have to know to develop an effective storage management system?

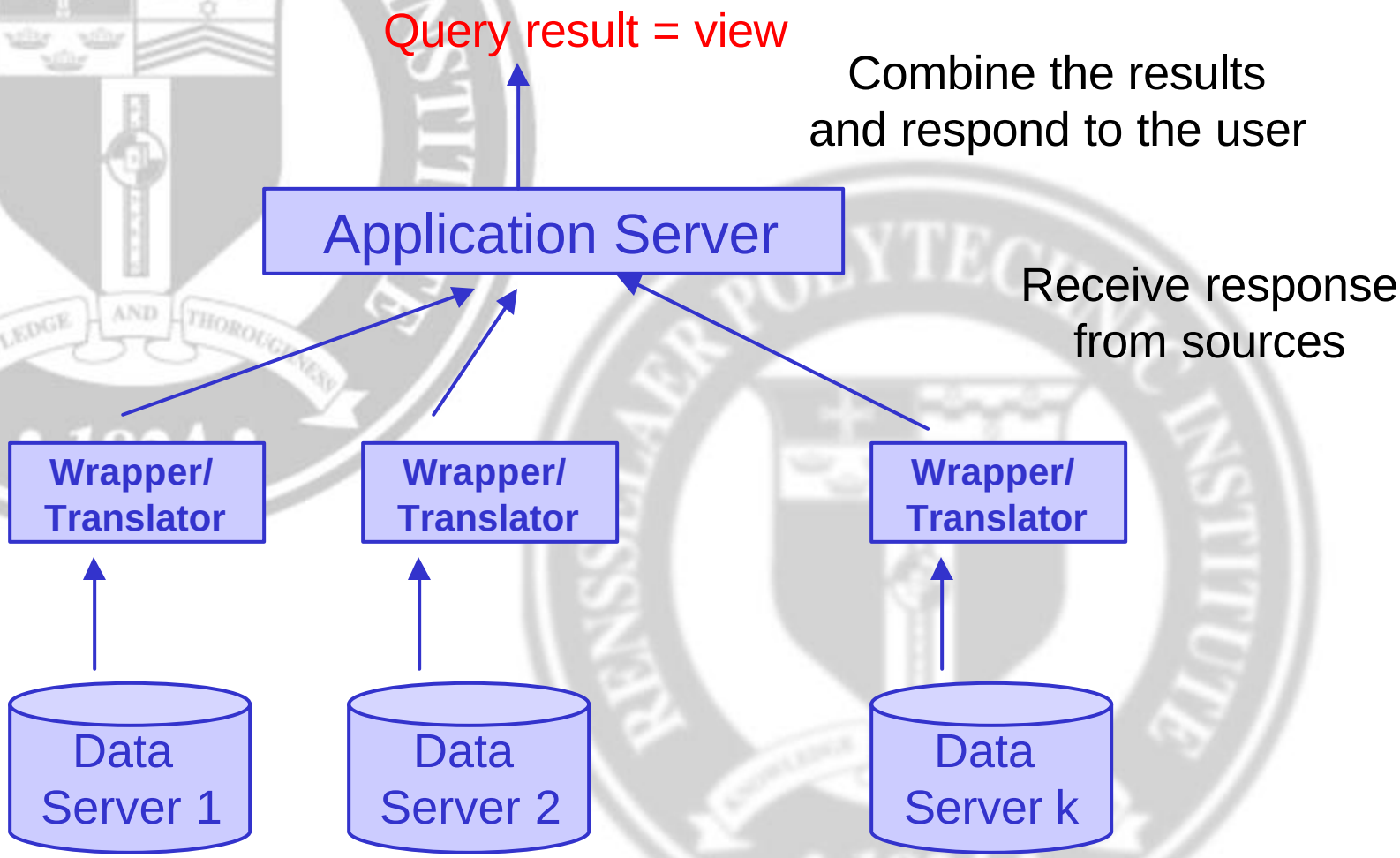
Networked applications

- Require data from distributed sources
  - Sources are distributed without the application in mind
- Sources are possibly autonomous, heterogeneous
  - No control over their internal data model
  - No agreements to notify the users of updates
- Sources are possibly integrated using views: local point of view of the data

Networked Applications



Networked Applications



Information integration

- Municipal network connecting multiple agencies to the office of the mayor
  - Population in different regions [CENSUS BUREAU]
  - Property boundaries [STATE TAX OFFICE]
  - Locations of traffic patterns, congestions [CITY]
  - Locations of major incidents [POLICE]
  - Dates/times of major events [LOCAL ORGANIZATIONS]

Information Integration

- Map the information of interest from each office to a "local schema"

  Population(Neighborhood, Popl, F%, M%, PC%, C%, H%, Ages0-20, Ages20-40, Ages40-60, Ages60-)

  Property(TaxID#, Neighborhood, Area, Boundary:Polygon)

  CaseReport(Day, Type, Neighborhood, TotalArrests, TotalCases)

Information Integration

- What each attribute means is defined by the "local schema"
- How to extract this information is defined the "integration scheme"
- Queries are posed to the application server that operates on the local schema
  - Which neighborhoods have the highest population of ages 20 and below per km^2?
  - Which neighborhoods have been showing increasing cases of pothole related murders?
  - What is the distribution of education across neighborhoods? [RESULT: map]

Information Integration

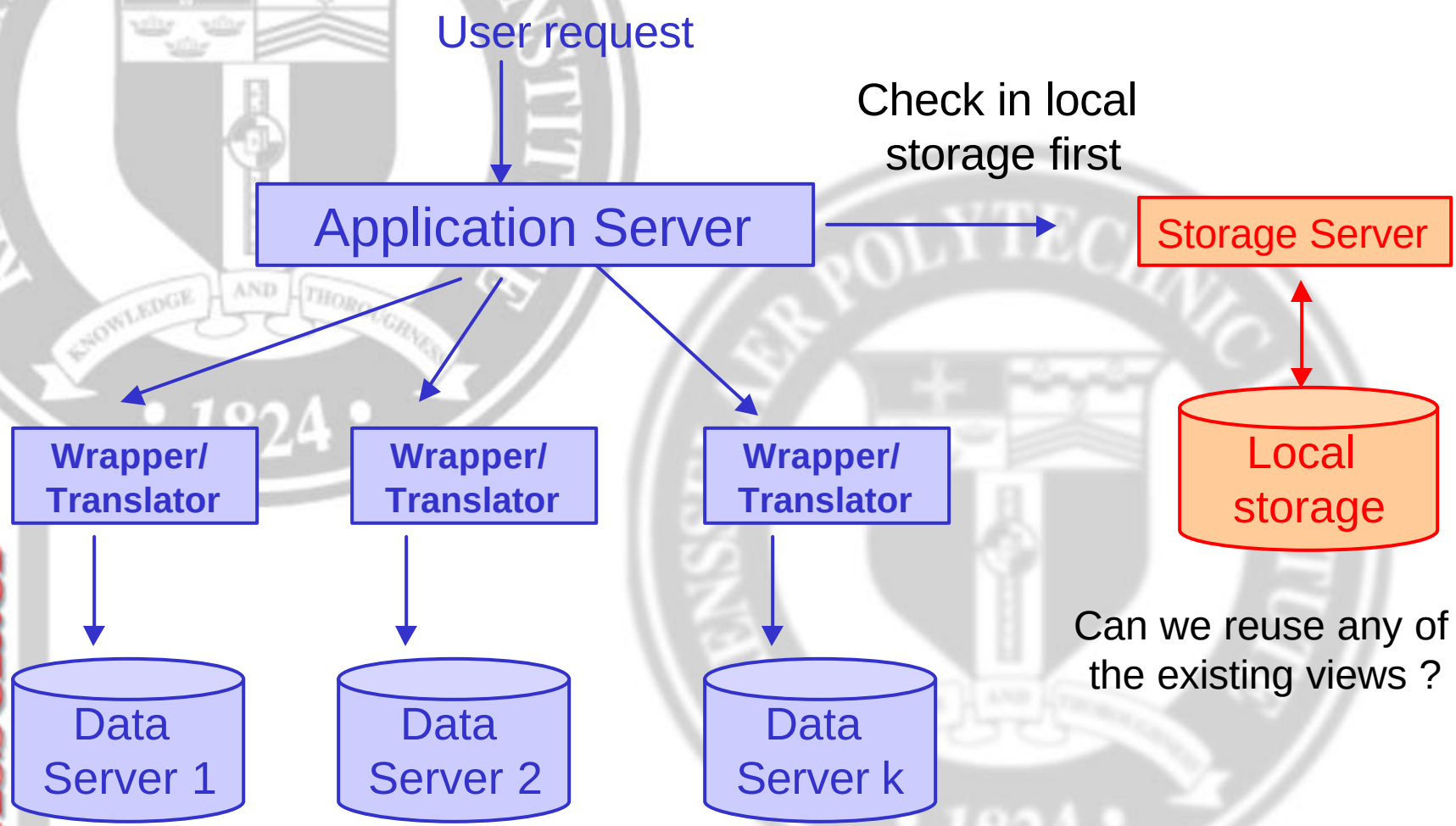
```
CREATE VIEW V1(A,B)
AS SELECT PT.neighborhood,
      sum(area(intersect(PT.boundary, MP.boundary)))
FROM property PT, maps MP
WHERE MP.neighborhood=PT.neighborhood
GROUP BY PT.neighborhood

CREATE VIEW V2(A,B,C,D)
AS SELECT C.day, C.neighborhood,
            avg(C.totalarrests), stddev(C.totalarrests)
FROM casereports C
GROUP BY cube(C.day, C.neighborhood)
```

Challenges

- Most data manipulation operations are very costly at the server
- Views are probably costly to transmit, too large to replicate
- New economies:
  - Unpredictable response time
  - Cost of accessing services /not cost of maintaining servers (pay-per-view)
  - Consistency is not a major concern→ archival data
  - Exact values are not always necessary → approximate answers

Networked Applications



Can we reuse any of the existing views ?

View reuse

- An existing view, fragment can be used to answer a new query:
  - Existing view: arrest data for the last six weeks
  - New query: arrest data for the last two weeks for 'South Troy'
  - Reuse: **select** on the existing view

  - Existing view: traffic congestion data for the last ten days for 'Niskayuna'
  - New query: traffic congestion summary for the last week for 'Niskayuna'
  - Reuse: **aggregate** on the existing view

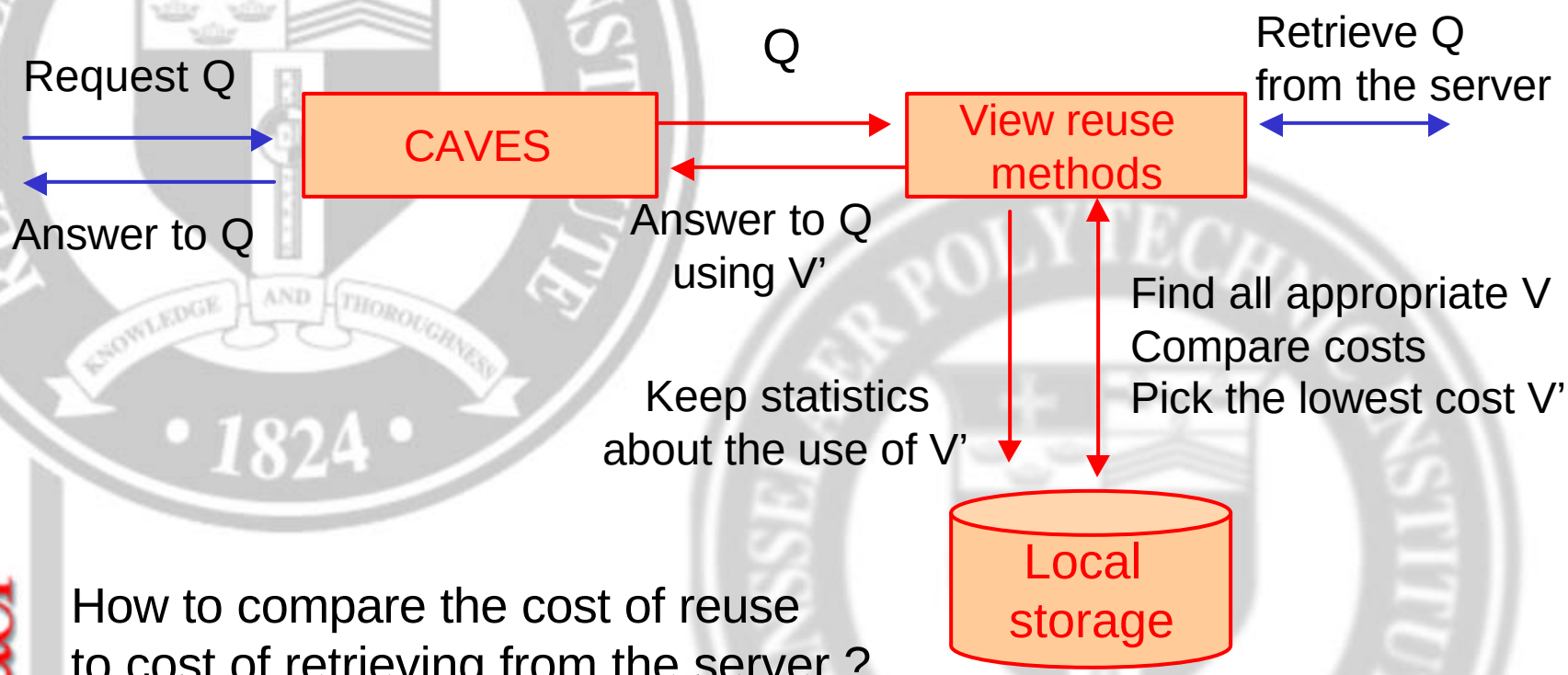
View reuse

- Existing: an image representing a map
- New: a smaller area completely contained within the existing map
- Reuse: crop the existing map/image

View reuse

- View reuse methods are being developed for all applications
  - Answering queries using views
  - Spatial indexing mechanisms
  - Each method is application specific, implemented by an AP and linked to CAVES
- Integrate view reuse methods with storage server
  - Instead of naïve look-up, invoke view reuse methods
  - Costly look-up methods may not always pay off → cost based

To reuse or not to reuse



How to compare the cost of reuse to cost of retrieving from the server ?

- Lower bound based on the expected response time and estimated size
- Always connect the server as a back-up plan

To reuse but…

- View consistency is a major concern for some applications
  - Keep a deadline for each "view type", and remove it after the deadline [our current approach]
  - Refresh views periodically by polling the servers [autonomous data servers]
    - Predict the rate of change based on observations
    - Invoke incremental view maintenance methods if they exist
  - Obtain leases on objects from servers, servers notify updates [dedicated data servers]

Statistics to keep

- For each view that is being stored, keep a number of statistics
  - Id, location on disk
  - Description → used to determine possible reuse
  - Performance statistics

Size
ComputationCost
ServerTransferRate
Age

LastRef
Hits
FrequencyOfUse
PercentageOfUse

Do not change by reuse

Change by reuse

Statistics

- Each statistics has a
  - Name
  - Method for initialization
  - <optional> Method for update

Incremented hits by 1
Poll servers

- Each application decides which statistics to keep
  - A number of statistics are provided by the system
  - They can be added by an AP

Storage Management

- Storage management is an optimization problem
  - A limited amount of storage space
  - Keep in storage items of highest value
- How to define the "value" of an item?
  - Use statistics to write functions that mimic the value objective → "order formula"
  - Use weights to determine the "importance" of an order formula in determining the value → "weight vector"
  - Weights represent a tunable parameter

Order formulae

Higher value represent higher priority → more desirable

F1 = computationCost/(size*age) [OLDESTFIRST]

F2 = hits [LU]

F3 = 1/serverTransferRate [FNSF]

F4 = lastRef [LRU]

F4 = max(k,hits)/(lastRef – lastToKRef) [LFU]

F5 = F1*F4 [LFUwSIZE]

Priority

- No order formula alone leads to better performance for all applications
  - Hits is not a good measure for items requested periodically
  - Computation cost and size alone are not good measures if they do not vary greatly among views
  - Computation cost may vary depending on the network and server load

Priority

- The system is usually biased against a new view with no hits
  - Use computation-cost/size for these views and then switch to the regular priority function [WATCHMAN]
  - Our solution:

    F1= computationCost/(size*age)

    F2 = hits

    Use: w1 * F1 + w2 * F2

Storage management with priority

- Given a set of order formula F1,..,Fk, a weight vector W=<w1,…,wk>, the priority of each view V is given by

  $$P(V) = w1 * F1(V) + \ldots + wk * Fk(V)$$

- If we need to remove a view to admit a new one, then we need to pick the one with the current lowest priority
- How can we do this quickly?
  - Keep all views in sorted order of their current priority using a priority queue

Storage management with priority

SS: total storage space

AvailableSpace = SS – sum{ size(V') | V' in storage }

V: new view just admitted to the cache

Pick views V1,…,Vk such that

P(V1)+…+P(Vk) is minimal over all views that free up size(V) – AvailableSpace

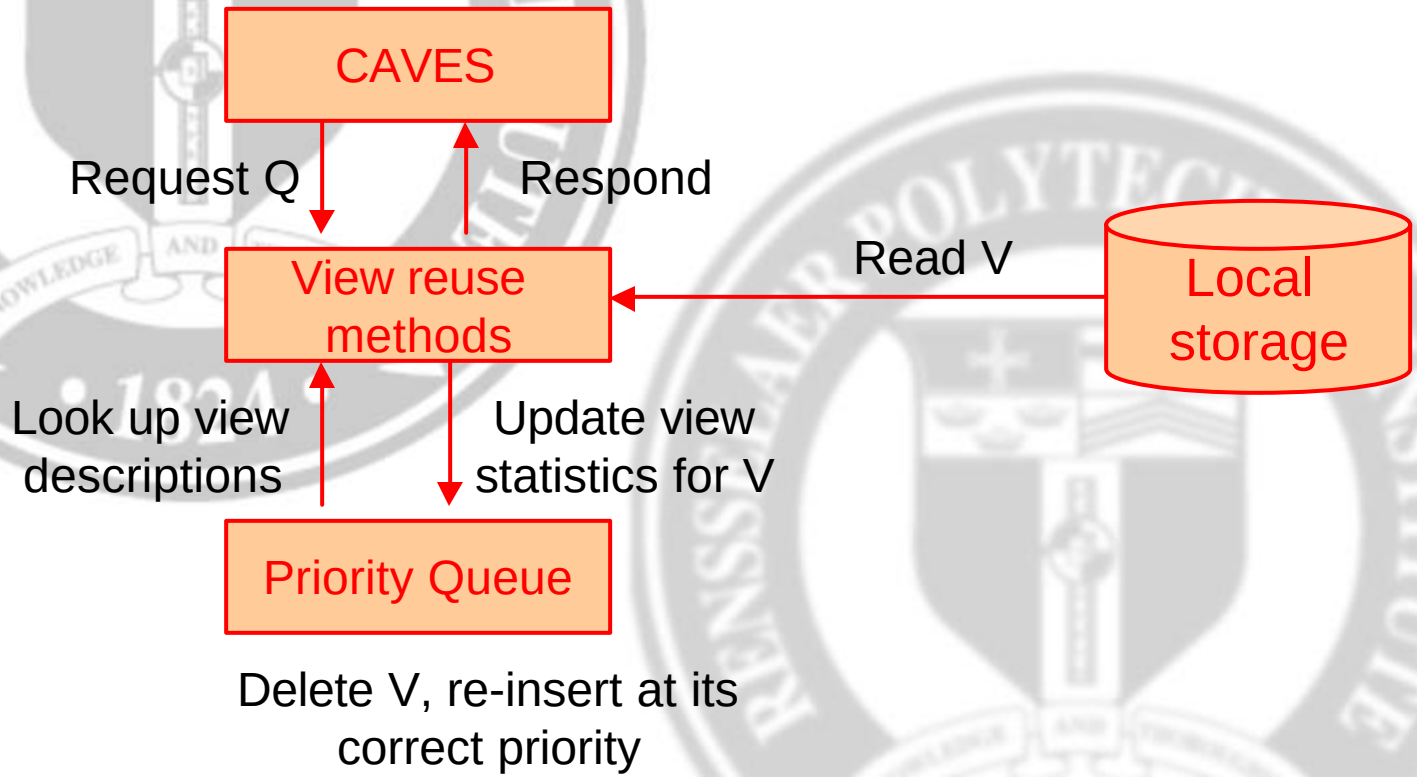
A version of the knapsack problem (NP-complete)

Storage management with priority

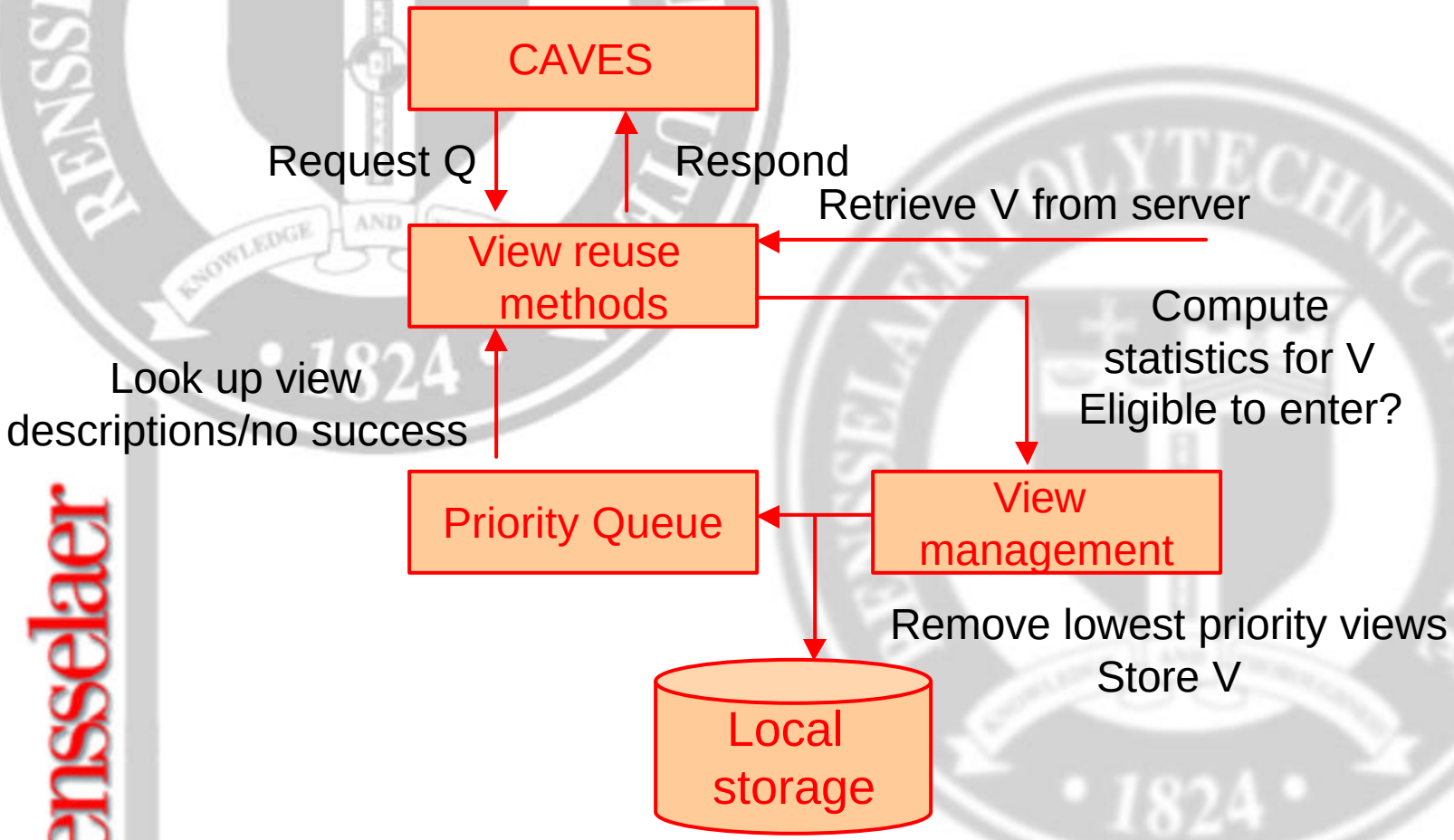
- Greedy heuristic:
  - Pick the items in the order of their priority until sufficient space is freed
  - Would like to choose bigger views first → smaller the number of views implies smaller total priority
  - Use unit priority instead of actual priority

    unitPriority = priority/sizeFactor
  - Small size factor → view size has no effect
  - Large size factor → priorities loose importance

Successful look-up



Unsuccessful look-up



Priority queue

- In this model, we update the priority queue only when a view is updated
- How about

  F1=computationCost/(size*age)
- Too costly to resort the queue regularly
- Keep two queues
  - Priority queue (PQ)
  - Timed output queue (TOQ) → schedule when the priority of certain views should be recalculated
  - For each view, create k ghosts in TOQ

Weight change rules

- It is unlikely that the same weights will maximize performance over all workloads
- Observe statistics over all order formulas in the system
  - Max, min, avg of values these functions take
  - Standard deviation of values in the PQ
- Use rules to define when the weights should change

  Stddev(F1)/(Max(F1)-Min(F1)) < .2 → [.2, F1]

Testing and Results

- The CAVES system is implemented in Java
  - Allows the specification of order formula and weight change rules from a system specification (XML document)
- A simulation model is implemented in C
- The simulation model and the actual system has been aligned for a number of test cases (30 min runs) on an Ultra10

Simulations

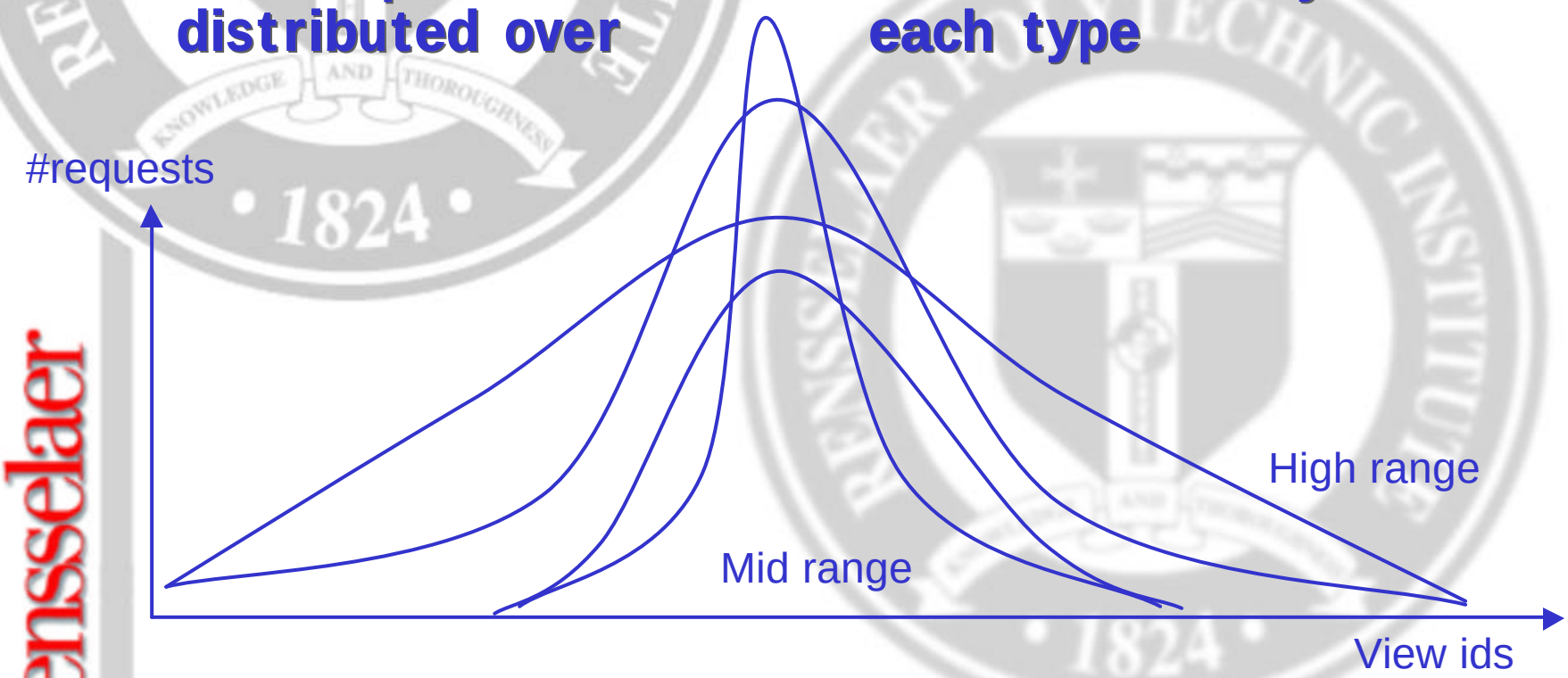
- Mixed workloads over five different query types
  - A type is defined by "size" of the resulting view, and the overall search time
  - An indexed selection is smaller in search time, varies in size [equality – range search], types 0,1,2
  - An aggregation query is very large in search time, typically small in size, types 3,4
  - Each type has a dedicated server, the network rate varies greatly for types 1 and 3.

Simulations

Qtype	# Desc	Size(K)	FF	CoC	W0	W1	W2	W3	W4	W5
0	10000	10-500	3-10	1-2	20	6	10	10	10	4
1	5000	175-200	90-100	2-2	20	47	5	40	10	2
2	5000	175-200	90-100	2-2	20	47	5	10	40	2
3	2000	10-50	50-100	2-3	20	0	40	40	40	2
4	1000	10-50	100-200	4-5	20	0	40	0	0	90

Simulations

- Tests with different state space of each query type [possible queries of a specific type] and the locality [the frequency of each requests]
- The requests are assumed to be normally distributed over each type



Simulations

- Functions f1=computationCost/(size*age), f2=hits, f3=1/nr
- Measured performance for runs of 10^5 seconds
- Varied a number of factors
  - Mean arrival time for requests [1 every 1,5,10 secs]
  - Mean time, standard deviation for network rate changes
  - Storage space
  - Weight combinations
- Measure Speedup = TimeWithoutCaves / TimeWithCaves

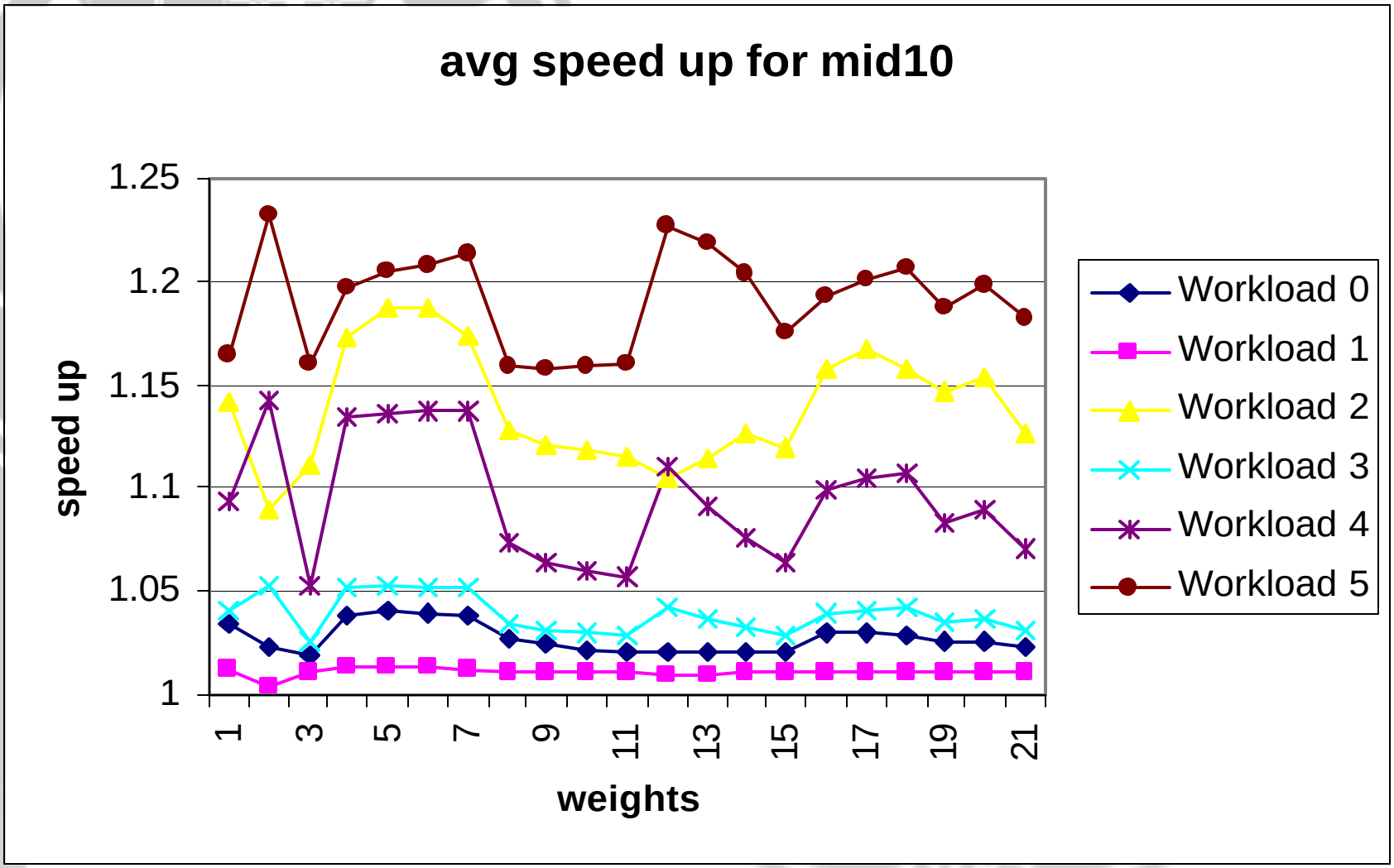
Results

- The optimal and near optimal weight cases vary from workload to workload, even for the same workload in different locality cases.
- The performance across cache sizes does not vary greatly across different weights cases.
- Average views in cache do not always predict performance.

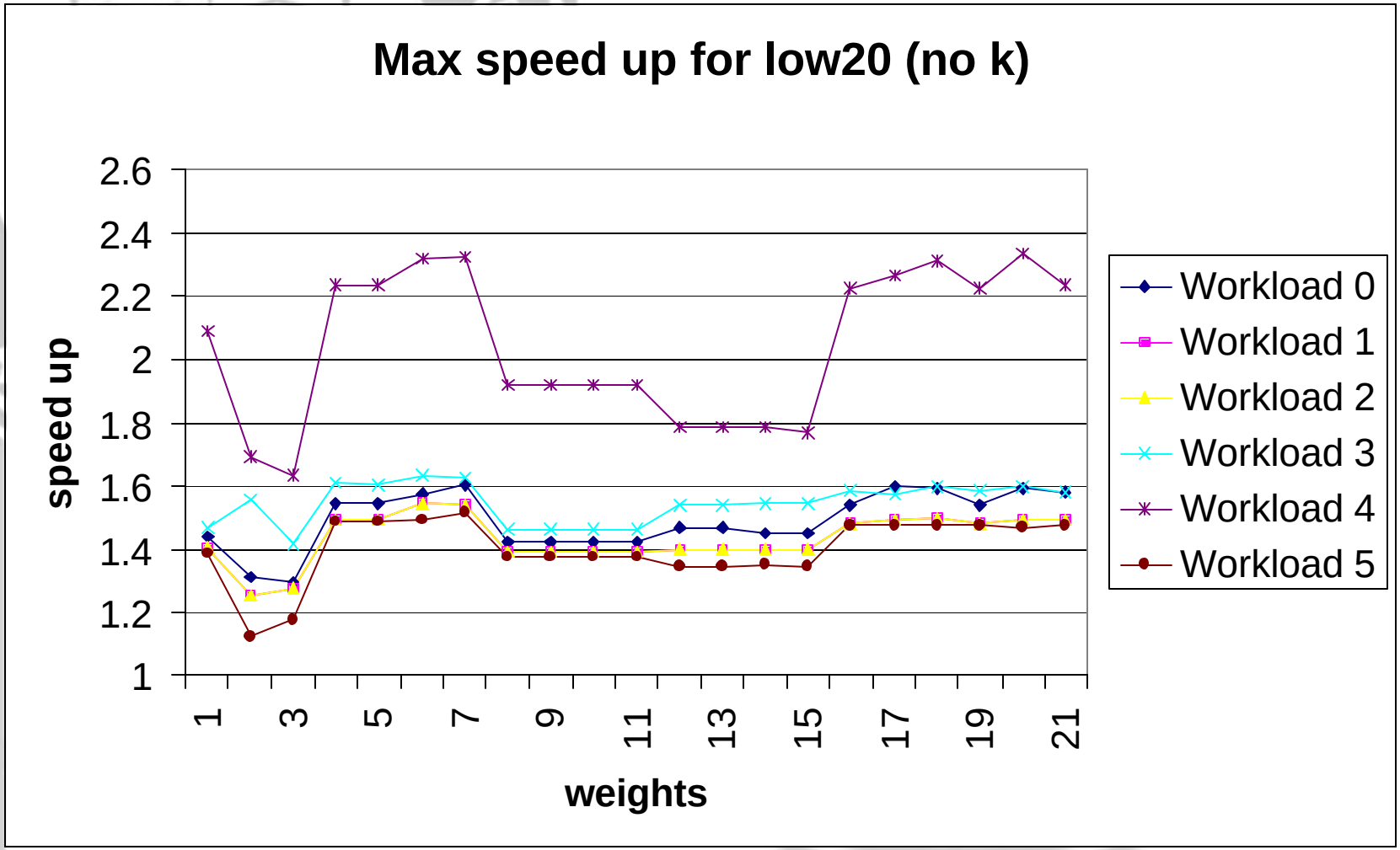
Results

- Functions:
  - Hit ratio (f2) predicts performance, but it does not always lead to the optimal performance.
  - Network rate (f3) is usually a factor in all optimal cases.
  - Size and computation cost (f1) is not always the best predictor of performance.
    - For workload 1, best performance always sets w1=0
    - For workload 5, best performance sets w1>0.

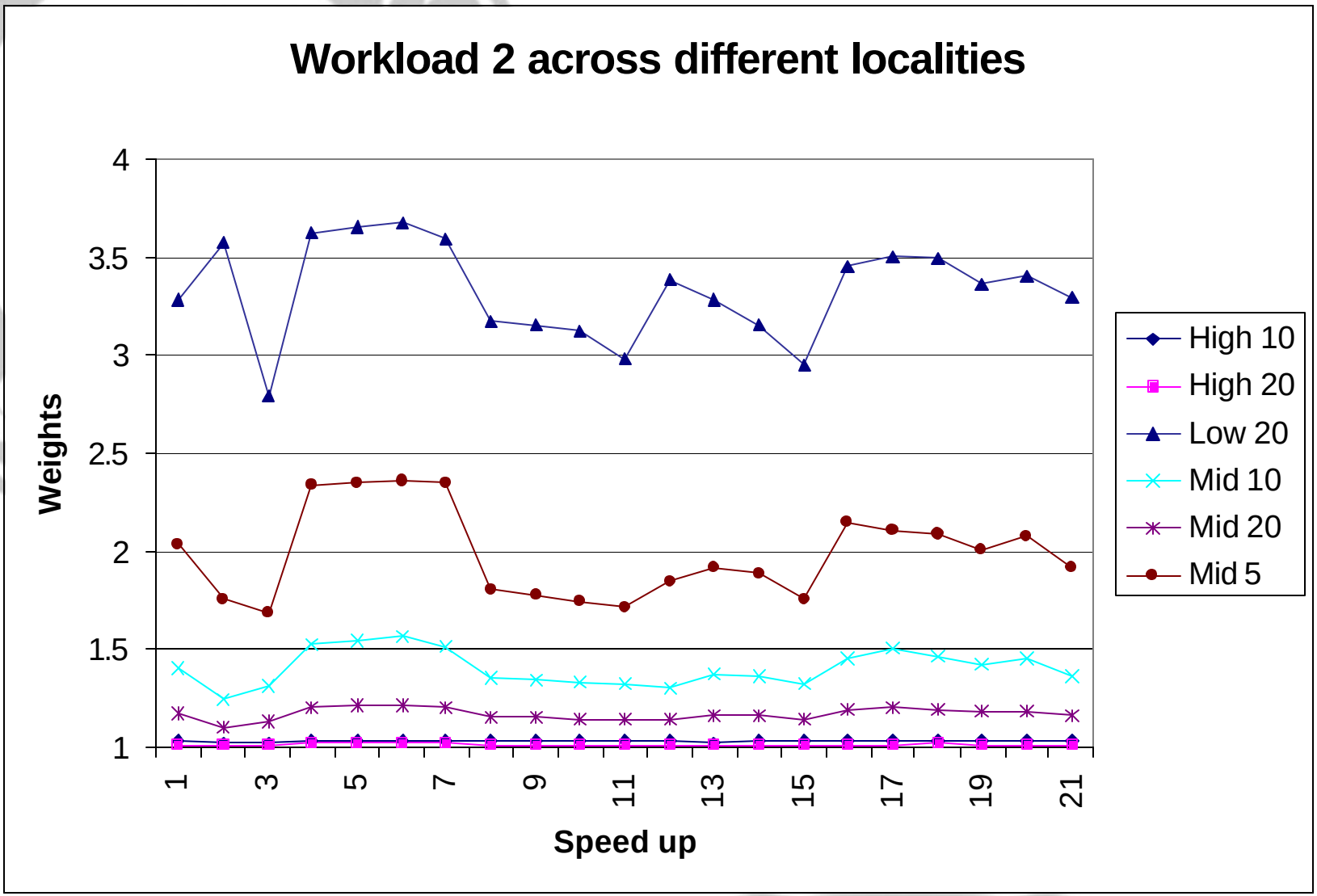
Weights: w1=<0,0,1> w2=<0,1,0> w3=<1,0,0> w4=<0,.2,.8> w5=<0,.4,.6> w6=<0,.6,.4> w7=<0,.8,.2> w8=<.2,0,.8> w9=<.4,0,.6> w10=<.6,0,.4> w11=<.8,0,.2> w12=<.2,.8 ,0> w13=<.4,.6,0> w14=<.6,.4,0> w15=<.8,.2,0> w16=<.2,.2,.6> w17=<.2,.4,.4> w18=<.2,.6,.2> w19=<.4,.2,.4> w20=<.4,.4,.2> w21=<.6,.2,.2>



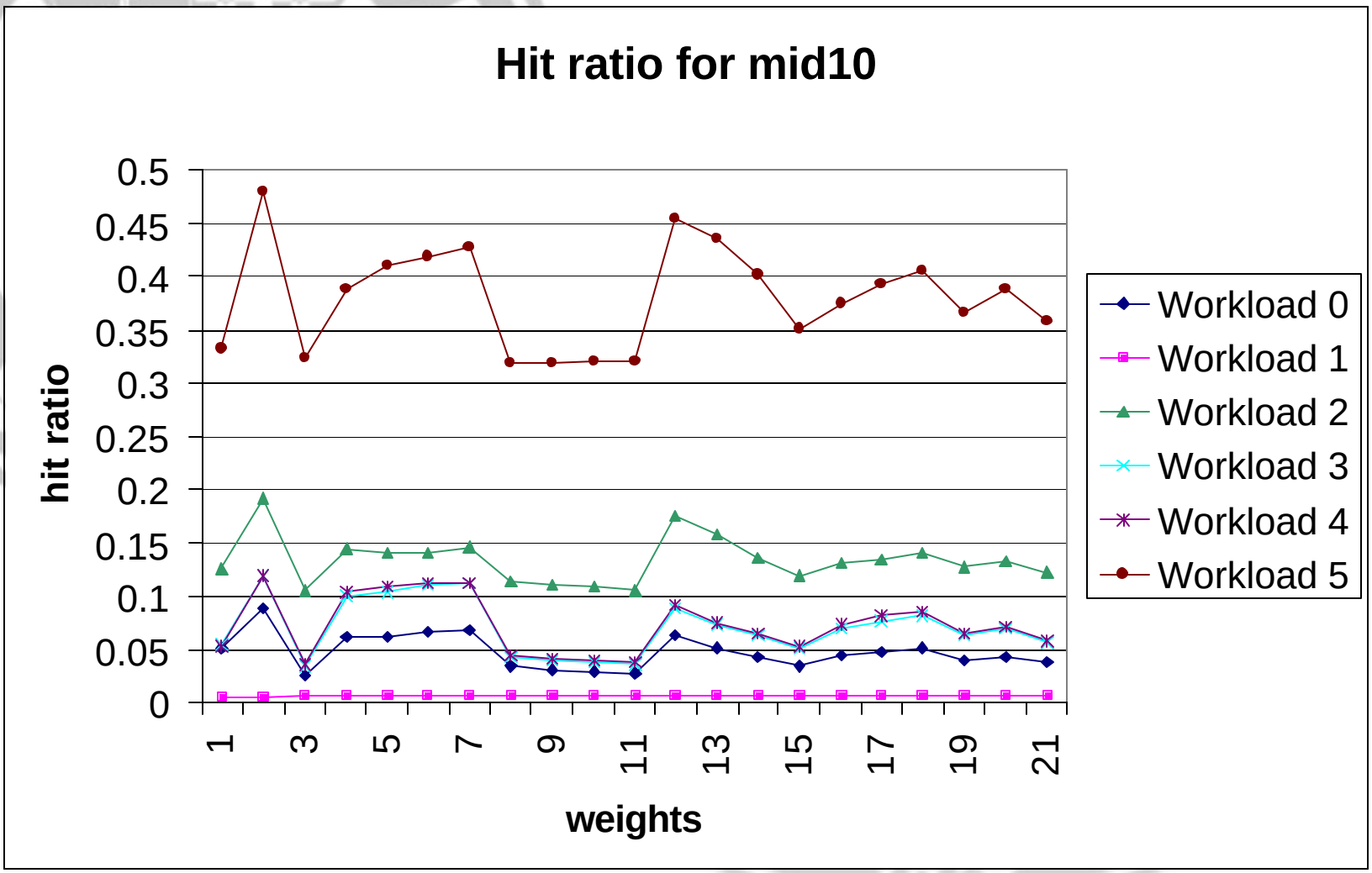
Weights: w1=<0,0,1> w2=<0,1,0> w3=<1,0,0> w4=<0,.2,.8> w5=<0,.4,.6> w6=<0,.6,.4> w7=<0,.8,.2> w8=<.2,0,.8> w9=<.4,0,.6> w10=<.6,0,.4> w11=<.8,0,.2> w12=<.2,.8,0> w13=<.4,.6,0> w14=<.6,.4,0> w15=<.8,.2,0> w16=<.2,.2,.6> w17=<.2,.4,.4> w18=<.2,.6,.2> w19=<.4,.2,.4> w20=<.4,.4,.2> w21=<.6,.2,.2>

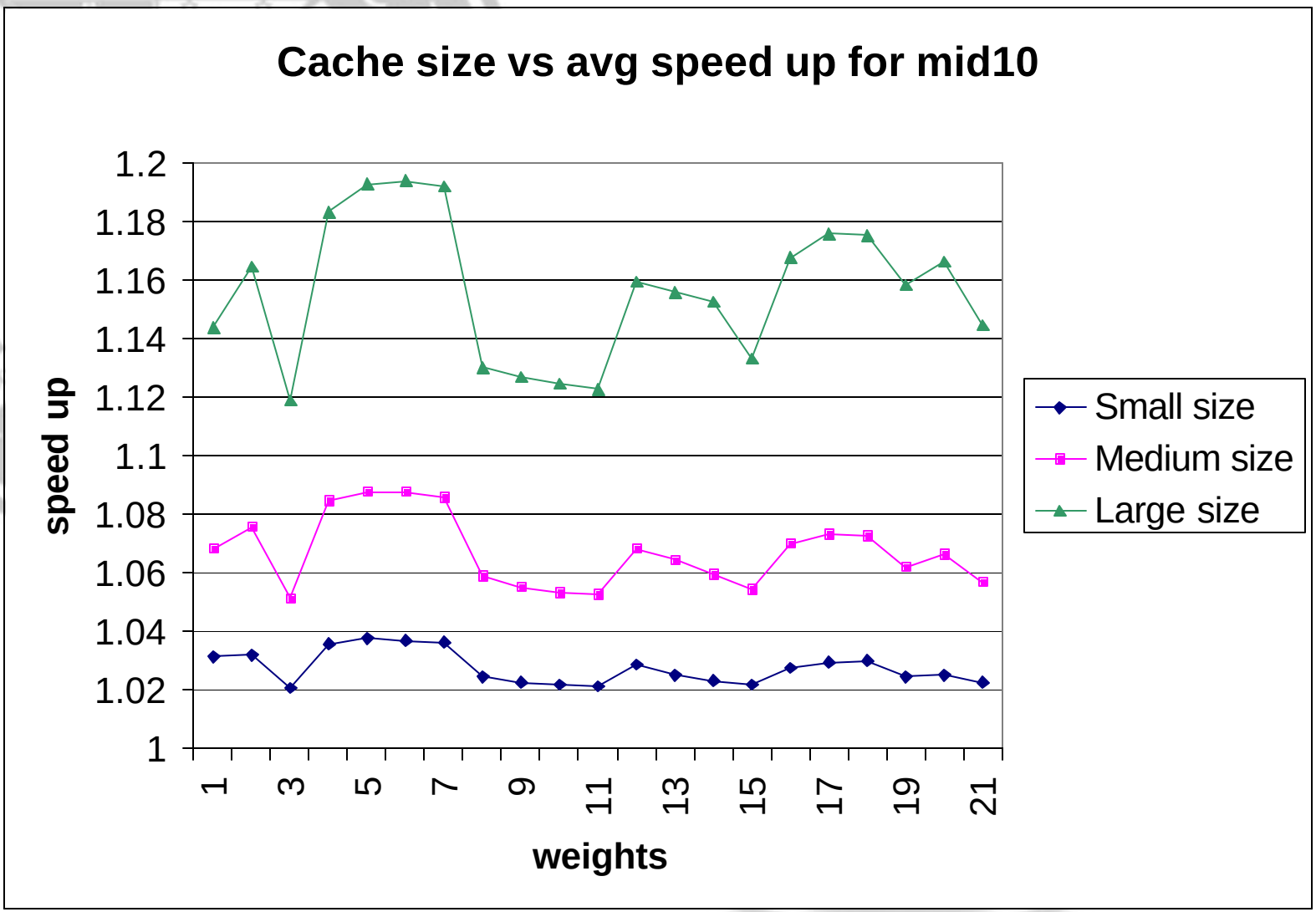


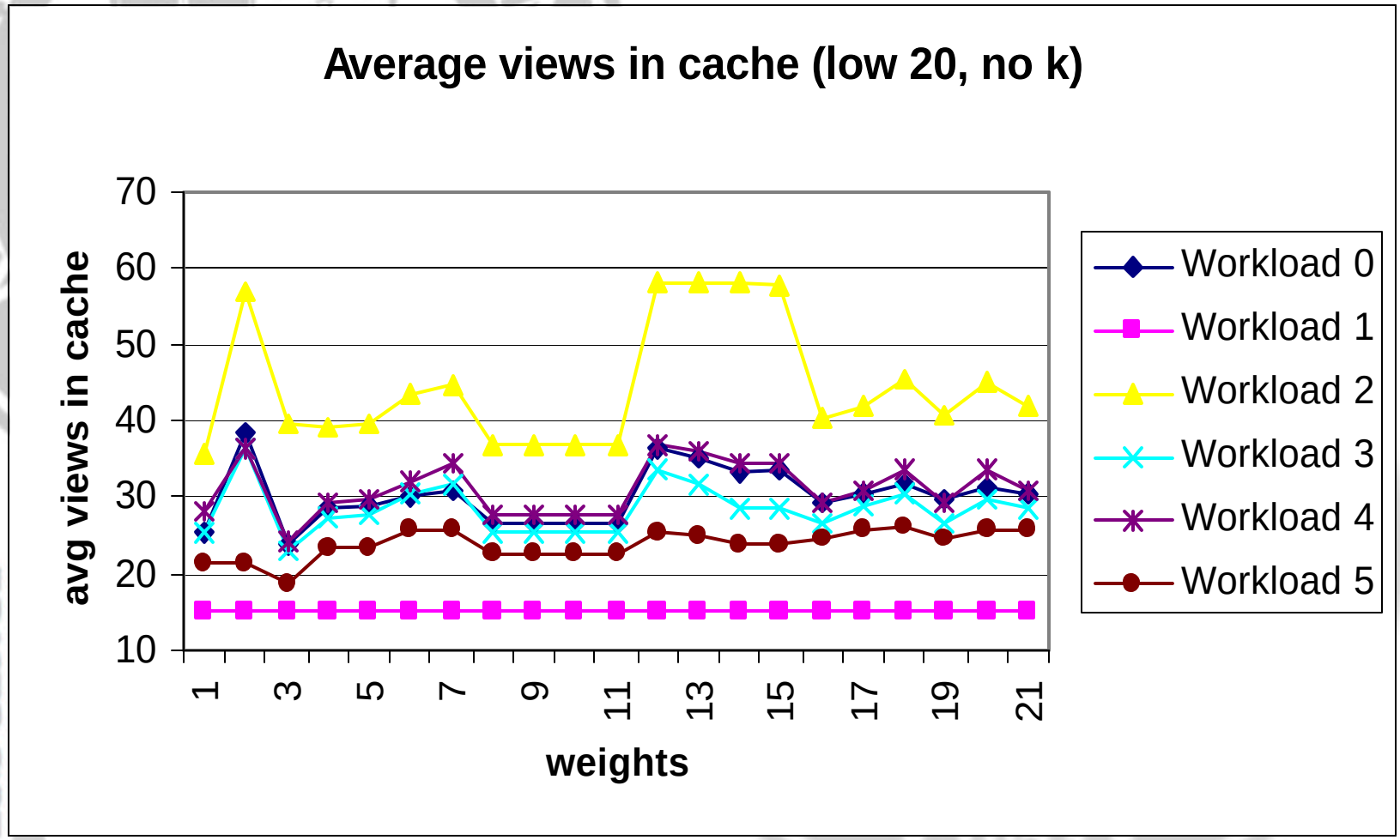
Weights: w1=<0,0,1> w2=<0,1,0> w3=<1,0,0> w4=<0,.2,.8> w5=<0,.4,.6> w6=<0,.6,.4> w7=<0,.8,.2> w8=<.2,0,.8> w9=<.4,0,.6> w10=<.6,0,.4> w11=<.8,0,.2> w12=<.2,.8 ,0> w13=<.4,.6,0> w14=<.6,.4,0> w15=<.8,.2,0> w16=<.2,.2,.6> w17=<.2,.4,.4> w18=<.2,.6,.2> w19=<.4,.2,.4> w20=<.4,.4,.2> w21=<.6,.2,.2>



Weights: w1=<0,0,1> w2=<0,1,0> w3=<1,0,0> w4=<0,.2,.8> w5=<0,.4,.6> w6=<0,.6,.4> w7=<0,.8,.2> w8=<.2,0,.8> w9=<.4,0,.6> w10=<.6,0,.4> w11=<.8,0,.2> w12=<.2,.8 ,0> w13=<.4,.6,0> w14=<.6,.4,0> w15=<.8,.2,0> w16=<.2,.2,.6> w17=<.2,.4,.4> w18=<.2,.6,.2> w19=<.4,.2,.4> w20=<.4,.4,.2> w21=<.6,.2,.2>

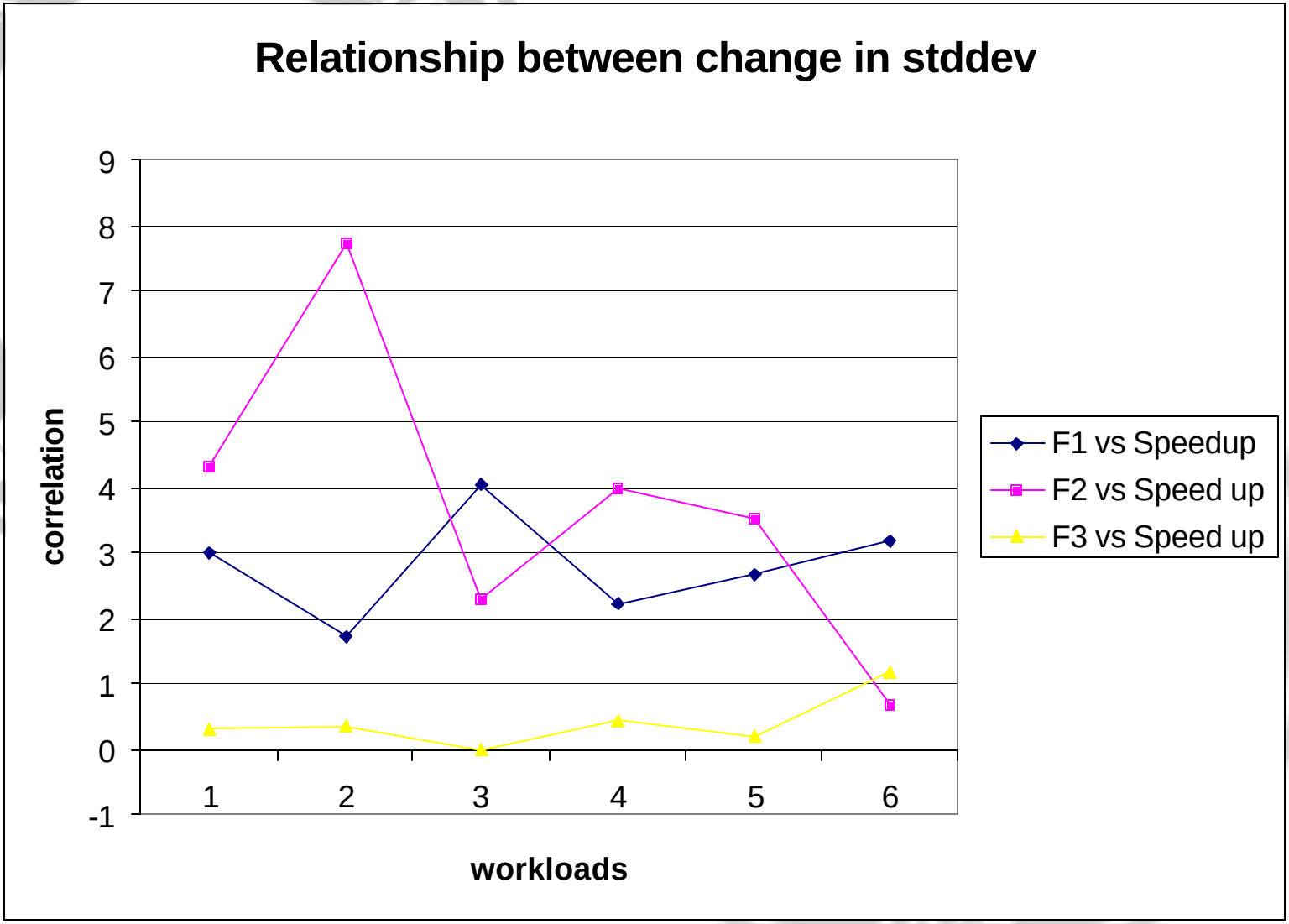



Cache size vs avg speed up for mid10
speed up
1.2
1.18
1.16
1.14
1.12
1.1
1.08
1.06
1.04
1.02
1
1
3
5
7
9
11
13
15
17
19
21
weights
Small size
Medium size
Large size


Average views in cache (low 20, no k)
avg views in cache
70
60
50
40
30
20
10
1
3
5
7
9
11
13
15
17
19
21
weights
Workload 0
Workload 1
Workload 2
Workload 3
Workload 4
Workload 5

Dynamic change rules

- If weights vary across all workloads, how to do we detect a workload and adjust the weight?
- A good formula should have a wide range of values and predict performance
  - ComputationCost = serverCost + transferCost
  - Change in the range of values may or may not be significant.


Relationship between change in stddev
correlation
9
8
7
6
5
4
3
2
1
0
-1
1
2
3
4
5
6
workloads
F1 vs Speedup
F2 vs Speed up
F3 vs Speed up

An Example - Mapping

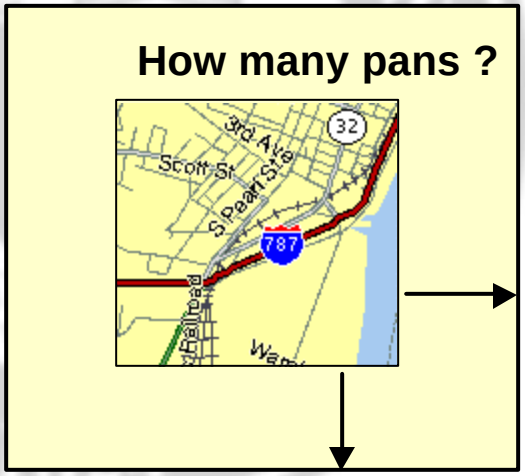
- Views = maps of specific types
- Reuse if a map can be cropped to satisfy the current request
- Support for map operations → Panning
  - If the user has requested a map, it is very likely that she will pan once in that map
  - If the user panned to left, it is more likely that she will pan to the same direction than the opposite direction

  [Learn and insert an actual model of user behavior here.]

An Example - Mapping

- New statistic: panFactor: how many pans are available to the last direction the user moved
- Order formula:
  - FOCUS: panFactor * last_use/now
  - Use a weighted sum

    w1 * FOCUS + w2 * F1 …



An Example - Mapping

- It is likely that the user is going to perform a zoom-in or a zoom-out operation on the same area next!
  - Prefetch what is likely to be useful in the future
  - View reuse method generates a set of "likely" queries Q1, …, Qk
  - Use a prefetch order formula to determine the priority of each Qi
  - Insert all Qi to a Prefetch Queue (PfQ)

Déjà vu?

- Prefetch queue operates exactly like priority queue
  - Available space (PQ) → available bandwidth (PfQ) [prefetch only when idle]
  - How to allocate bandwidth? → call the highest priority views
  - Optimize → Savings in the overall expected delay
  - How to define priority?
    - Focus based on likelihood
    - Complexity → based on map complexity

An Example - Mapping

- Highly complex views will save most in terms of delay, but will keep the server busy
- What size to get?
  - A larger view is more likely to be useful now and in the future
  - A smaller view will incur smaller overall cost in case of a mistake
- What entry criteria to use?
  - Theoretically, a generated request is less likely to be repeated than an actual (human generated) request

Conclusions

- CAVES provides a general purpose storage management system that is programmable and adaptive.
- Research funded by NSF
- Group members

S. Adali
C. Carothers
D. Spooner
G. Yaun
C. Chiang

Research topics

- Determining semantic overlap of views is too costly in practice
  - Approximate measures
  - Computed by view reuse methods incrementally
  - Extend the PQ management function to include a dependency graph and cost measures
- Semantic caching [Franklin, Godfrey]
- Data Cube optimizations [Harinayan, Ullman]

Research Topics

Use simulation as a decision module using actual load characterizations to find

- Optimal/near optimal weight distributions
- Useful measures that predict performance
- Discover useful weight change rules

Probabilistic models for network loads

Tests with actual data sets

Research topics

- Scalability: networked or hierarchical storage servers [Kalnis Papadias]
  - Each server has its own priority queue and priority functions
  - Objectives vary at each level
    - The priority of a view depends on the priority of views in other servers
    - Each server observes a different request distribution, aggregated at higher levels