Lawrence Berkeley Laboratory
UNIVERSITY OF CALIFORNIA, BERKELEY

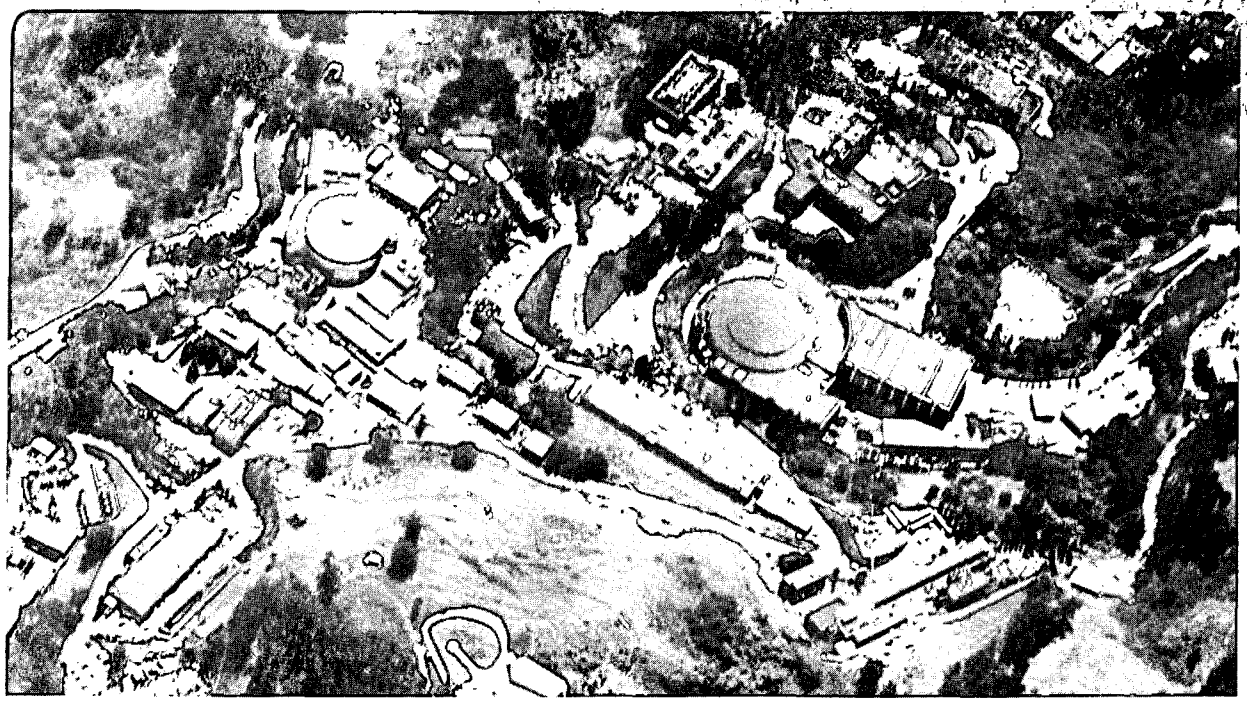
Information and Computing Sciences Division

Presented at the 7th Annual Symposium on Principals of Database Systems, Austin, Texas, March 21–23, 1988

Analytical Modeling of Materialized View Maintenance Algorithms

J. Srivastava and D. Rotem

October 1987



Prepared for the U.S. Department of Energy under Contract Number DE-AC03-76SF00098.

DISCLAIMER

LBL-24137

Analytical Modeling of Materialized View Maintenance Algorithms

Jaideep Srivastava
Computer Science Division
University of California
Berkeley, CA 94720

Doron Rotem
Computer Science Research Department
Information & Computing Sciences Division
Lawrence Berkeley Laboratory
University of California
Berkeley, CA 94720

October 1987

Presented at the 7th Annual Symposium on Principals of Database Systems, March 21-23, 1988, Austin, Texas

---

This research was supported by the Applied Mathematics Research Program of the Office of Energy Research, U.S. Department of Energy under Contract DE-AC03-76SF00098.

Analytical Modeling of Materialized View Maintenance Algorithms

Jaideep Srivastava

Computer Science Division
University of California
Berkeley, CA 94720

Doron Rotem

Computer Science & Mathematics Dept.
Lawrence Berkeley Laboratories
University of California
Berkeley, CA

1. Introduction

A materialized view is a stored copy of the result of retrieving the view from the database. We consider here, views that can be constructed from the relational algebra operations *select, project* and *join*. Also aggregates such as *sum* or *count* over views are considered.

Materializing a view before a query is made on it has been a recent proposal. Conventional systems use query modification, where the query on a view is modified to operate on one or more of the base relations [STON 75]. Being derived from the base relations, materialized views have duplicated data. Any changes in the latter have to be reflected in the former, and *vice versa*, to maintain consistency between them. Refreshing materialized views by generating periodic *database snapshots* has been proposed by [ADIB 80, LIND 86]. [BLAK 86] has proposed the *immediate refresh policy*, i.e. updating the copy of the view after each transaction. [ROSS 86, HANS 87] have proposed the *deferred refresh policy* in which the view is updated just before data is retrieved from it. [BUNE 79] presented a method for analyzing each update command before execution to see if it could cause a view change. If the system could not rule out the possibility, the view would be completely recomputed. This was done for evaluating complex trigger and

alerter conditions.

Performance of view materialization algorithms can be significantly improved by *screening* of each tuple inserted into or deleted from the base relations. If a tuple fails the test then it cannot change the view, and hence does not cause a view refresh. The screening described in [BLAK 86] is done by substituting the tuple in the view predicate, which is then tested for satisfiability. Carrying out this test for each tuple incurs significant runtime cost. [BUNE 79] proposes a screening test which has a compile time phase and a runtime phase. During the compilation of a transaction a check is made to see if any of the fields being updated are present in a view definition. If no such fields exist, the update is a *readily ignorable update* (RIU) with respect to the view, and cannot cause it to change. If the transaction is not an RIU, the individual tuples are screened further at runtime. With RIU transactions there is associated only a per-transaction cost, while with non-RIU ones the cost associated is proportional to the number of tuples involved. This is similar to the cost pattern of [BLAK 86]'s screening scheme. A scheme called *rule indexing* has been proposed by [STON 86]. Index intervals covered by one or more clauses of the view predicate are locked by special markers called *t-locks*. When a tuple is inserted into the relation, if an index record containing a t-lock is disturbed, the tuple passes the screening test. This can produce *false drops* (i.e. tuples which pass the screening test but do not satisfy the view predicate), which are handled by substituting the tuple in the predicate.

Research in the maintenance of materialized views has focussed on two independent problems. First is the problem of screening tuples, either at compile-time or at run-time, to decide if an update needs to be done to the view at all. Second is the problem of update frequency of the materialized view that maintains it up-to-date, and incurs low cost. [BUNE 79, BLAK 86] have focussed on the first problem while [ADIB 80, LIND 86, ROSS 86, HANS 87] have focussed on the second. [BLAK 86] also outlines a scheme for the second problem. In this paper we develop a queueing theory based analytical model for view materialization which focusses on the second problem. A class of efficient update algorithms are presented that are shown to be more general than those proposed earlier. As far as we know this is the first attempt to develop an anlytical model for this problem.

Section 2 outlines the existing algorithms for materialized view maintenance. Section 3 introduces a queueing theory model for the materialized view maintenance problem and

discusses the criteria on which the algorithms will be evaluated. Section 4 presents our algorithms for two variations of the problem and discusses their generality. Finally, Section 5 presents conclusions and suggests directions for future research. for view materialization

2. Analytical Modeling of View Materialization

This section describes the mechanics of maintaining materialized views, develops a queueing model for it, and discusses the criteria for comparative evaluation of maintenance algorithms.

2.1. Mechanics of Materialized View Maintenance

A materialized view is a stored copy of the view which is created at the time of its definition. Any changes made to the base relations have to be reflected in the materialized view. This is done by means of a periodic maintenance process or *refresh process*. The mechanics of materialized view maintenance can be described in terms of the files that exist in the system and the processes that manipulate them.

Files: The base relations from which the view is derived are stored in the file *BaseRel* and the materialized view in the file *MatView*. If MatView is refreshed as soon as a change is made to BaseRel, then these are the only files required. However, deferring the refreshing of MatView has advantages of being more efficient under certain conditions, [ROSS 86, HANS 87]. Thus another file, the *TempFile*, is required to store the changes between successive refreshes to the MatView. These are shown in *Fig. 5.3(a)*.

Processes: There are three kinds of processes in the system which are of interest to us, as shown in *Fig. 5.3(a)*. First are Read Queries, or *Reads*, that are directed to the MatView and processed using it. Second are the Update Queries, or *Updatess*, that can be directed either to the MatView or to the BaseRel. These are handled by making the appropriate changes to the BaseRel † and also recording it in the TempFile. If the Update is an insert (delete), a data item is added to (deleted from) the BaseRel and the change is

† Changes to BaseRel are done anyways and *not* because of maintaining materialized views.

recorded in the TempFile. Handling Updates directed to BaseRel requires more care. Only some of these Updates affect MatView and are the only ones that have to be accounted for. This is done by *screening* each Update directed at the BaseRel against a *filter* (i.e. the logical predicate defining the view), to determine if it does indeed affect the view [BUNE 79, BLAK 86]. The ones that do so are recorded in the TempFile in addition to being added to (deleted from) the BaseRel. The third kind of processes are the Refreshes, or *Ref*s, which are executed periodically and whose function is to refresh the MatView using the contents of TempFile and bring it up-to-date with respect to BaseRel. This involves inserting/ deleting all data items that have been marked for insertion/ deletion respectively in TempFile. A point to note is that no changes have to be made to BaseRel when a Ref is executed, since the former is always up-to-date. Also, every time a Ref is executed, it merges MatView and TempFile, emptying the latter.

2.2. A Queueing Model

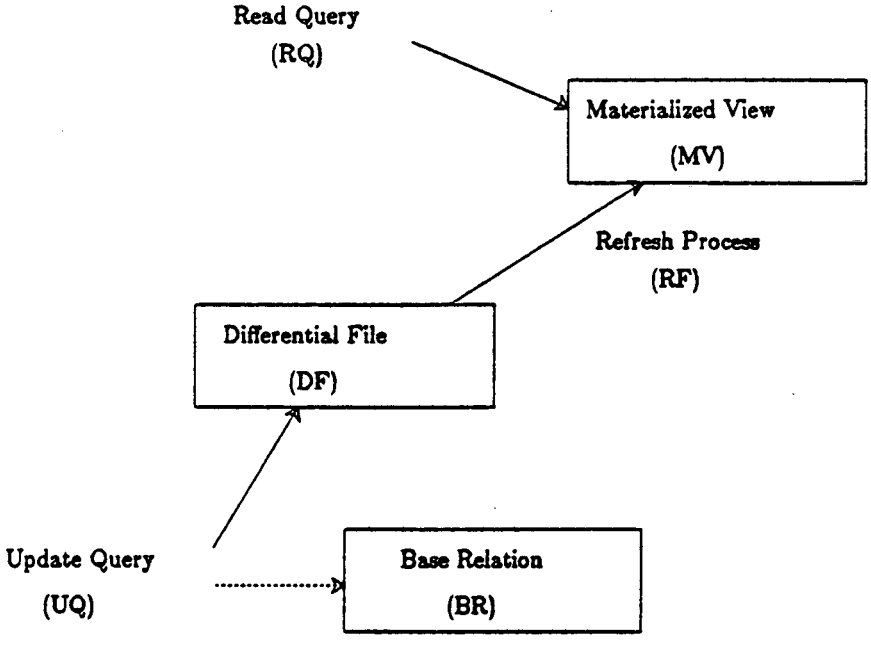
The operation of the materialized view maintenance mechanism is modeled as a *queueing system*. The arrival of a Read or an Update, or the creation of a Ref is the arrival of a job for service, and the time required for its execution is the service time. *Fig. 5.3(b)* gives a pictorial representation of the model.

Nature of the Processes: The Reads and Updates are both assumed to be stochastic processes having the Poisson distribution [ROSS 85]. Their arrival rates are λ_R and λ_U respectively. Thus,

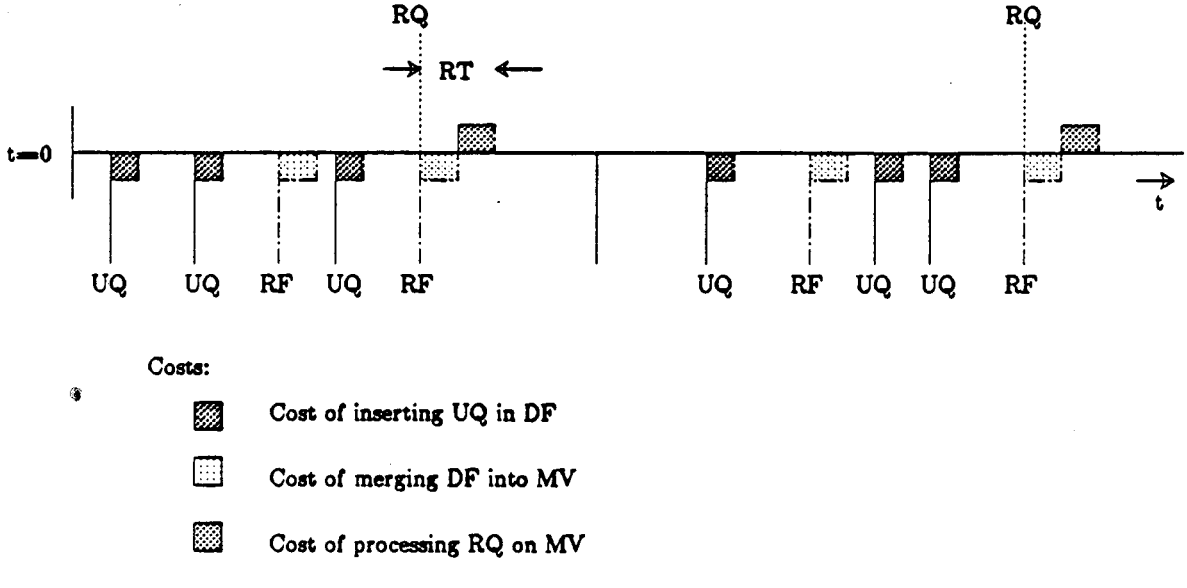
$$Read \sim Poisson\ (\lambda_R).$$

$$Update \sim Poisson\ (\lambda_U).$$

Costs associated with processes: The cost associated with a process is its execution time. For a fair comparison of materialized view maintenance algorithms we have to develop a cost model that measures precisely the *extra overhead* that a database system incurs in maintaining MatView and TempFile. An example is the effort required to handle insertions/ deletions to BaseRel. However, this would be required even if there were no materialized views, and thus we do not include its cost. Thus, the cost of a process includes precisely the extra effort it has to do for view maintenance. There is a cost associated with *screening* the tuple to decide if it affects the view; and if it does, there is the



(a)



(b)

Fig. 5.3 *Mechanics of Materialized View Maintenance.*

additional cost of inserting it in the DF. Given below are the costs associated with the three processes in our model.

C_R: The cost of accessing the required data from MatView, and processing it to answer the query.

C_U: If the update is to MatView, the only cost is of recording it in TempFile. When the update is to BaseRel, it also has to be screened to see if it affects MatView. From our point of view only the *relevant updates* [BLAK 86], i.e. the ones that affect the view, are of interest. Since screening is done in main memory, its cost is negligible and is henceforth ignored. Thus, the cost associated with a Update is that of recording it in TempFile regardless of whether it is directed to MatView or BaseRel.

C_F: The cost of merging TempFile with MatView to bring the latter up-to-date.

The work done in maintaining the files MatView and TempFile is expressly for the purpose of view materialization. Thus the extra cost incurred by the database system shold be borne by the queries made to the view. This is done by taking the total extra work done in a certain time period, dividing it by the number of view queries in the same period, and adding the result to the cost of processing each view query. Table 5.2 lists the various parameters of the model.

2.3. Evaluation Criteria

The performance of materialized view maintenance algorithms can be seen from two viewpoints, namely the user's and the system's. The only concern of the former is the minimization of the *response time* of his query, i.e. the time that elapses between the submission of a query and the return of the result. The only concern of the latter is the minimization of the resources spent in doing the overall job. None is concerened with the other's objective, and this may lead to conflicting requirements for view maintenance algorithms. We define below two evaluation criteria, the first captures the user's viewpoint while the second the system's.

AWC: Average Waiting Cost, i.e. the average time elapsed between the submission of a query and the returning of results.

APC: Average Processing Cost, i.e. the amount of effort spent by the system, on the average, for processing a query. This includes the cost of manipulating the BaseRel as well as the average cost of the extra work done for materialized view maintenence.

Symbol	Meaning
Read	Read Query
Update	Update Query
Ref	Refresh Process
p_R, p_U, p_F	probabilities of arrival
$\lambda_R, \lambda_U, \lambda_F$ $\lambda_j = \lambda p_j$	arrival rates of processes
G_R, G_U, G_F	service distribution of processes
C_R, C_U, C_F	service time of processes
$\rho_j = \lambda_j E(C_j)$	utilization of process j
$G = \sum p_j G_j$	overall service distribution
C	overall service time
$\rho = \sum \rho_j = \lambda E(S)$	overall utilization
Q_j	time average number of *j-jobs* in queue
$Q = \sum Q_j$	time average number of jobs in queue
d_j	average queue delay of a *j-job*
$d = \sum p_j d_j$	average queue delay of a job

Table 5.2 *Queueing Notation Used.*

A linear combination of the above, called Average Query Cost (or *AQC*), is a generalized criteria and can be expressed as,

$$AQC = \alpha(APC) + (1-\alpha)AWC; \qquad 0 \leq \alpha \leq 1$$

α and $1-\alpha$ are importance attached to the user and system respectively.

3. Materialized View Maintenance Strategies

4. Efficient View Maintenance

This sections discusses the materialized view maintenance problem, keeping in mind both the user's and the system's viewpoints. Two versions of the problem are discussed. The first does not include queueing delays while the second does. Update policis are designed which minimize the Average Query Cost (AQC), which is a combination of the Average Waiting Cost (AWC) and the Average Processing Cost (APC).

Consistency Requirement A requirement that has not been explicitly stated so far is that whenever a Read is processed the MatView should be up-to-date, i.e. the effect of all the Updates preceding the Read must be reflected in it. Relaxing this requirement, i.e. allowing the data in the materialized view to become outdated or *imprecise*, introduces a whole new dimension to the problem [SRIV 87].

4.1. Problem 1: Queueing Delays are Negligible; Algorithm A1

The problem is to determine how often should a Refresh process be run, i.e. how often should the MatView be brought up-to-date by merging the contents of the TempFile with it. The stochastic nature of the arrivals of Reads and Updates causes some queueing delays, which for now are considered negligible. Previous research [ADIB 80, LIND 86, BLAK 86, HANS 87] also ignores queueing effects.

The aim is to design the Refresh process, *Ref*, that minimizes the Average Query Cost (AQC). Thus, formally

Given,

$$Read \sim Poisson\ (\lambda_R).$$

$$Update \sim Poisson\ (\lambda_U).$$

Design a refresh process Ref that minimizes the objective function,

$$\alpha APC + (1 - \alpha)AWC.$$

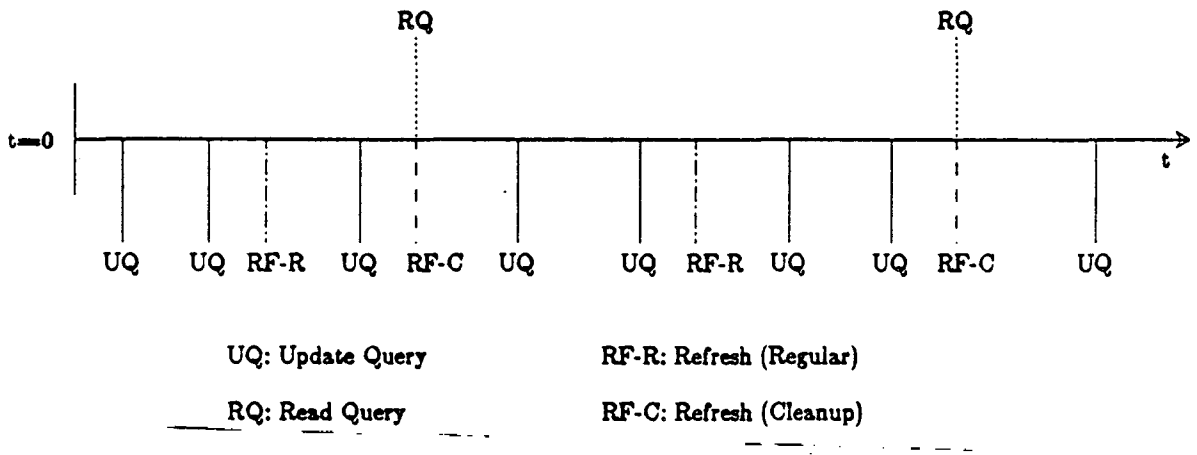


Fig. 6.2 *Optimal Refresh Process, Ref_{opt}.*

The approach we take is to design the optimal refresh process, Ref, as a *superimposition* of two processes, refresh-regular (Ref-R) and refresh-cleanup (Ref-C) Thus,

$$\text{Ref} \equiv \text{Ref-R} \oplus \text{Ref-C}$$

where $\oplus$ is the superimposition operator. *Fig. 6.2* shows our design. This two part design is necessitated by the consistency requirement mentioned above. Howsoever the process Ref may be designed, unless it is identical to the process Read, it is possible for MatView not to be up-to-date because of arrivals of Updates since the last execution of a Ref. Whenever a Read arrives, an instance of Ref has to be generated to take care of these residual Updates. As shown in *Fig. 6.2* the Refs forcibly generated on the arrivals of Reads are called Ref-Cs. The rest of the Refs are called Ref-Rs. Since there exists exactly one instance of Ref-C for each instance of Read, we have,

$$Ref\text{-}C \sim Poisson\ (\lambda_R).$$

Now we are left with the problem of designing the process Ref-R to optimize the objective function, and then design Ref by superimposing it with Ref-C ($\equiv$ Read). The process that would achieve optimality could be some general process. However, it would in general be extremely difficult to characterize it and then superimpose it on Ref-C. Thus, we restrict our search of the optimal Ref-R to the set of Poisson processes, because of the useful property that the superimposition of any number of Poisson processes is itself Poisson.

The actual design of the process Ref-R depends on the various parameters of BaseRel and MatView. The parameters we choose are taken from [HANS 87], wherein he has compared his and [BLAK 86]'s algorithms. This choice of parameters, shown in *Table 6.2*, will enable us to compare our algorithm with theirs.

Symbol	Meaning
N	# of records in *BaseRel*
f	size of MatView as a fraction of BaseRel
T	# records per disk block
C_d	cost of a disk access
$n=1$	# records inserted/ deleted by a Update
C_U^{A1}	Cost of processing an Update using algorithm A1
C_R^{A1}	Cost of processing a Read using algorithm A1
C_F^{A1}	Cost of processing a Ref using algorithm A1

Table 6.2 *Database Parameters Used.*

Consider*Fig. 6.2*, where our design of the optimal Ref is shown. TempFile is merged into MatView whenever an instance of Ref occurs, and each Update inserts/ deletes a

single record,

$$E[\textit{size of TempFile}] = (\textit{arrival rate of Update})^*(E[\textit{interarrival time of Ref}])^*1$$

$$\Longleftrightarrow E[\textit{size of TempFile}] = (\lambda_U)^*(\frac{1}{\lambda_F}) = \frac{\lambda_U}{\lambda_F}$$

$$C_U^{A1} = C_d{}^*y(\frac{\lambda_U}{\lambda_F}, \frac{\lambda_U}{T\lambda_F}, 1)\dagger$$

Also,

$$E[\#\ \textit{of Updates between succ Refs}]$$

$$= (\textit{arrival rate of Update})^*(E[\textit{interarrival time of Ref}])$$

$$= \frac{\lambda_U}{\lambda_F}$$

$$\Longleftrightarrow C_F^{A1} = C_d{}^*y(fN, \frac{fN}{T}, \frac{\lambda_U}{\lambda_F})$$

We assume that in the steady state the net size of MatView does not change appreciably over time, i.e. the rates of insertion and deletion to it are roughly the same. For analysis purposes we henceforth consider the size of MatView to be a constant over time and equal to fN records. This assumption is the same as made by [HANS 87]. Hence the cost of processing a Read on MatView, i.e. C_R^{A1}, is constant.

Now,

$$APC^{A1} = C_R^{A1} + C_F^{A1}{}^*(E[\#\ \textit{of Ref arrivals between succ Reads}])$$

$$+ C_U^{A1}{}^*(E[\#\ \textit{of Update arrivals between succ Reads}])$$

$$\Longleftrightarrow APC^{A1} = C_R^{A1} + C_F^{A1}{}^*(\frac{\lambda_F}{\lambda_R})$$

$$+ C_U^{A1}{}^*(\frac{\lambda_U}{\lambda_R})$$

$$AWC^{A1} = C_R^{A1} + C_F^{A1}$$

The final step in the solution is to consider the expression Average Query Cost (AQC^{A1})

$$AQC^{A1} = \alpha APC^{A1} + (1-\alpha)APC^{A1}$$

which has only a single independent variable, λ_F, and minimize it using a standard minimization technique like calculus.

† $y(n,m,k)$ is the expected number of disk accesses required to retrieve a group of k records together, from a database havig n records and m data blocks.

4.2. Special Cases of Algorithm A1

In this section we show that our algorithm is more general than those proposed earlier. Specifically, we show that the algorithms proposed by [ADIB 80, LIND 86, BLAK 86, ROUS 86, HANS 87] are special cases of our algorithm.

Lemma 6.3 *DRA is a special case of A1.*

Proof: On making $\alpha = 1$, the general cost equation above becomes,

$$AQC^{A1} = APC^{A1}$$

$$= C_R^{A1} + (\frac{\lambda_U}{\lambda_R}) * C_d * y(\frac{\lambda_U}{\lambda_F}, T\lambda_F, 1)$$

$$+ (\frac{\lambda_F}{\lambda_R}) * C_d * y(fN, \frac{fN}{T}, \frac{\lambda_U}{\lambda_F})$$

The Yao function, $y(n,m,k)$, is *n-Monotone Decreasing* as well as *m-Monotone Decreasing*, as mentioned in Appendix A. Hence, the above expression is minimized for the smallest possible value of λ_{RF}. Now, since $Ref = Ref-R \oplus Ref-C$,

$$\lambda_F = \lambda_{F-R} + \lambda_{F-C}$$

And,

$$Ref-C \equiv Read$$

$$\Longleftrightarrow \lambda_R = \lambda_{F-R} + \lambda_R$$

Hence, AQC^{A1} is minimized when

$$\lambda_{F-R} = 0$$

$$\Longleftrightarrow Ref \equiv Read$$

The algorithms proposed by [ROUS 86, HANS 87] are DRAs. Thus we conclude that they are special cases of algorithm A1.

Lemma 6.4: *IRA is a special case of A1.*

Proof: On making $\alpha = 0$ the general cost equation becomes,

$$AQC^{A1} = AWC^{A1}$$

$$= C_R^{A1} + C_d * y(fN, \frac{fN}{T}, \frac{\lambda_U}{\lambda_F})$$

The Yao function is *k-Monotone Increasing*, as mentioned in Appendix A. Hence, the above expression is minimized for the smallest possible value of k, i.e $k = 1$.

$$\Longrightarrow \frac{\lambda_U}{\lambda_F} = 1$$

$$isRef \equiv Update$$

The algorithms proposed by [ADIB 80, LIND 86, BLAK 86] are all IRAs. Thus we conclude that they are special cases of algorithms A1. The above two lemmas are equivalent to proving the following theorem and corollary.

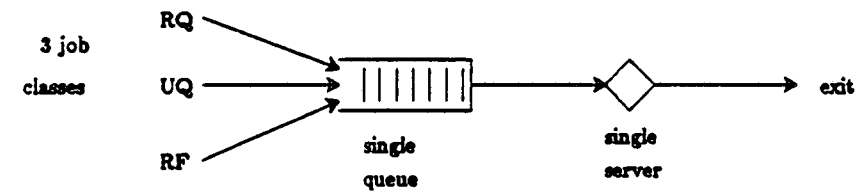
Theorem 6.2: *DRA and IRA are special cases of A1.*

Corollary 6.1: *When queueing delays are negligible,*

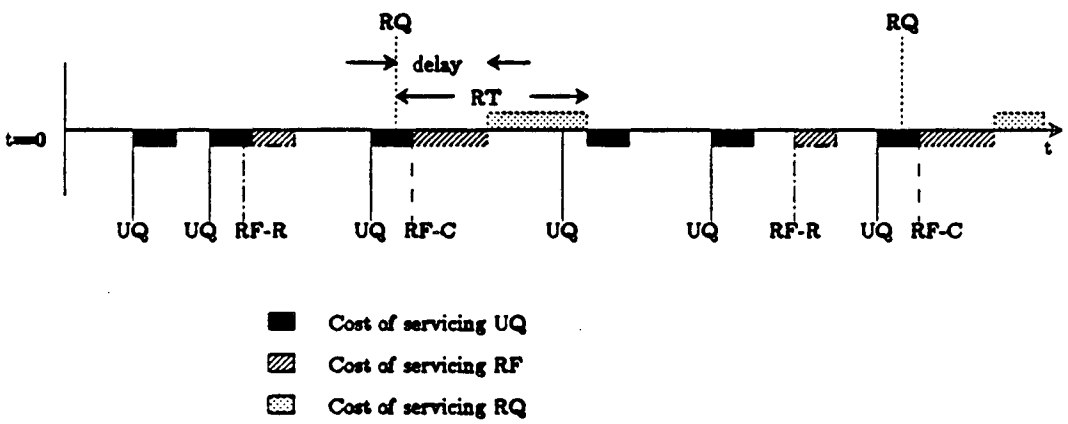
(i) DRA is an optimal algorithm if only APC is considered.

(ii) IRA is an optimal algorithm if only AWC is considered.

4.3. Problem 2: Queueing Delays are Significant; Algorithm A2



(a)



(b)

Fig. 6.3 *Queueing System Model.*

Fig. 6.3(a) shows a queueing system that models the view maintenance mechanism. The characteristics of the system are as follows:

Arrival Distribution: There are three classes of jobs, Read, Update and Ref. Their arrivals are Markovian † with arrivals rates λ_R, λ_U, and λ_F respectively.

† Read and Update are given as Markovian. As discussed in previous sections, the requirement of strict consistency makes the choice of Ref as Markovian an attractive one too.

Service Distribution: The service distributions are G_R, G_U, and G_F.

Server: There is a single server in the system.

Queue Discipline: FIFO, except Ref has higher priority than Read. This ensures that Refs are serviced before Reads, to ensure strict consistency.

Rest of the parameters remain the same as Problem 1. Thus, formally

Solve Problem 1 taking queueing delays into consideration.

The average processing cost for a query, APC, is not affected by queueing considerations because it depends only on the size of the database. The average response time for queries is affected though, as shown in *Fig. 6.3(b)*. Now,

$$\lambda = \lambda_R + \lambda_U + \lambda_F$$

$$p_R = \frac{\lambda_R}{\lambda}$$

$$p_U = \frac{\lambda_U}{\lambda}$$

$$p_F = \frac{\lambda_F}{\lambda}$$

Let g_R, g_U, g_F be density functions of the three service times C_R, C_U, C_F, respectively. Now,

$$g_R\text{: } P(S_R = C_R) = 1$$

$$g_U\text{: } P(S_U = y(\frac{\lambda_U}{\lambda_F}, \frac{\lambda_U}{T\lambda_F}, 1)) = 1$$

$$g_F\text{: } P(S_F = y(fN, \frac{fN}{T}, k))$$

$$= P(\Lambda(t+\frac{1}{\lambda_F}) - \Lambda(t) = k)$$

$$= \frac{e^{-\frac{\lambda_U}{\lambda_F}}}{k!} * \left[\frac{\lambda_U}{\lambda_F}\right]^k$$

Also, if g is the density function for the overall service time, C. Then,

$$g = \sum p_j g_j$$

The average delay of a Read job, d_R, is

$$d_R = \frac{\lambda E(S^2)}{2(1-\rho_F)(1-\rho_F-\rho_R)}$$

The Average Response Time of a query, AWC^{A2} consists of two components, the expected service time of the query, C_R^{A2}, and its average queueing delay, d_R.

$$isa AWC^{A2} = d_R + C_R^{A2}$$

The average processing cost of the query, APC^{A2} is the same as before because it depends only on the database parameters and not on queueing delays. The last step in designing the optimal refresh process is to determine the value of λ_F that minimizes the expression,

$$AQC^{A2} = \alpha APC^{A2} + (1-\alpha)AWC^{A2}$$

Note: Part (ii) of corollary 6.1 may not hold any more. The intuitive reason for this being that the triangle-inequality property of the Yao function favors grouping Update jobs before service because the overall service time is reduced, thus reducing queueing delay.

5. Conclusions

In the recent past there has been increasing interest in the idea of maintaining materialized copies of views, and use them to process view queries [ADIB 80, LIND 86, BLAK 86, ROSS 86, HANS 87]. Various algorithms have been proposed, and their performance analyzed. However, there does not exist a comprehensive analytcal framework under which the problem can be systematically studied. We present a queueing model which faclitates both a systematic study of the problem, and provides a means to compare various proposed algorithms. Specifically, we propose a parametrized approach in which both the user and system viewpoints are integrated, and the setting of the parameter decides the relative importance of each. *Table 5.1* below compares our work with that done in the past.

Algorithm	APC	AWC	Mathematical Model	Remarks
Adiba 80	-	-	-	heuristic
Lindsay 86	-	-	-	"
Blakeley 86	-	X	-	optimizes on AWC w/o queueing considerations; author doesn't mention
Roussopoulos 86	X	-	-	optimizes on APC; author doesn't mention
Hanson 87	X	-	-	intuitive argument to optimize on APC
Srivastava 87	X	X	X	model using renewal & queueing theories; optimization based on APC and AWC

Table 5.1 *Comparison of various algorithms.*

We have described one kind of mechanics for materialized view maintenance. Many variations on the basic theme are possible. One is to use the same TempFile to store the updates for more than one materialized view. Whenever one of the views needs to be refreshed, the others are refreshed automatically. The advantage comes from the reduced overhead of maintaining one TempFile instead of several. Another variation is to use idle CPU and disk time time to refresh views so as to reduce the work when a view query actually arrives. Such an approach is reminiscient of Dijkstra's concurrent *on-the-fly garbage collection* [DIJK 7]. The analysis of these variations is a non-trivial task. The solutions presented above select the refresh process from the set of Poisson processes. Other variations would be to consider preiodic time and periodic count policies. A comparative study of these policies would be interesting.

Materialized views have their most potential in a distributed environment where a copy of the view is kept at the user's site [LIND 86], or in a workstation-based database environment where each user has a workstation with a copy of the view while the main database resides at a remote location [ROSS 86]. It has been shown that materialized views work very well in such environments. The analytical model presented here can be enhanced by introducing communication costs, i.e. modeling the communication channnel an message queues as part of an overall queueing network. This can be used to analyze the performance of materialized view maintenence algorithms in a distributed environment.

6. *References*

[ADIB 80] Adiba, M. and B.G. Lindsay, "Database Snapshots", Proc. of the Intl. Conf. on VLDB, October 1980, pp86-91.

[AGRA 83] Agrawal, R. and D.J. DeWitt, "Updating Hypothetical Data Bases", Information Processing Letters 16 (April 1983), 145-146, North Holland.

[BLAK 86] Blakeley, J.A., P.Larson and F.W.Tompa, "Efficiently Updating Materialized Views", Proc. of the 1986 ACM-SIGMOD Conf. on Management of Data, Washington DC, May 1986, 61-71.

[BUNE 79] Buneman, O.P. and E.K. Clemons, "Efficiently Monitoring Relational Databases", ACM Transactions on Database Systems 4,3 (September 1979).

[CARD 75] Cardenas, A.F., "Analysis and Performance of Inverted Database Structures", Comm. of the ACM 18, 5 , May 2975, 253-263.

[HANS 87] Hanson, Eric N. "A Performance Analysis of View Materialization Strategies," Proc. of the 1987 ACM-SIGMOD Intl. Conf. on the Management of Data, San Francisco, CA, May 1987.

[LIND 86] Lindsay, B.G., L.Haas, C.Mohan, H.Pirahesh, and P.Wilms, "A Snapshot Differential Refresh Algorithm", Proc. of the 1986 ACM-SIGMOD Conf. on the Management of Data, Washington DC, May 1986, 53-86.

[ROUS 86] Roussopoulos, N. and H.Kang, "Principles and Techniques in the Design of ADMS+/-", Computer, December 1986.

[SRIV 87] Srivastava, J. "Precision-Time Tradeoffs: Data Management and Query Processing in Distributed Databases", Qualifying Exam Proposal, CS Division, University of California, Berkeley, June 1987.

[STON 75] Stonebraker, M., "Implementation of Integrity Constraints and Views by Query Modification", Proceedings of the 1975 ACM-SIGMOD International Conference on Management of Data, San Jose, CA, June 1975.

[STON 86] Stinebraker, M., T. Sellis and E. Hanson, "An Analysis of Rule Indexing Implementations in Data Base Systems", Proceedings of the First Annual Conference on Expert Database Systems, Charleston SC, April 1986.

[WOOD 83] Woodfill, J. and M. Stonebraker, "An Implementation of Hypothetical Relations," Proc. of the 9th VLDB Conference, Florence Italy, December 1983.

[YAO 77] Yao, S.B., "Approximating Block Accesses in Database Organizations," CACM 20, 4 April 1977.

LAWRENCE BERKELEY LABORATORY
TECHNICAL INFORMATION DEPARTMENT
UNIVERSITY OF CALIFORNIA
BERKELEY, CALIFORNIA 94720