20.1.1

Given relation R is distributed evenly,

a) p = 8, t = (100ms * 100)/8 = 1.25s
b) p = 100, t = (100ms * 100)/100 = 100ms
c) p = 1000, t = (100ms * 100)/1000 = 10ms

20.1.2

To avoid extra disk I/O's,

(M-1) > B(S)/p, given B(S) < B(R)

20.2.1

The map function takes an input document d, goes through the document character by character, and each time it encounters another word w, it emits the pair (w, i), where i is the id of the document d. The intermediate result is a list of pairs (w_1, i_1),(w_2, i_2),….
The input to reduce function is a pair $(w, [i_1, i_2, …, i_n])$, where the i's are document ID's. The reduce function takes a list of ID's, eliminates duplicates, counts the occurrence of the unique ID's, and produces the count.

20.2.2

a) The map function takes a tuple t and emits the pair (v, i), where v is the value under c and i is the id of the tuple t. The intermediate result is a list of pairs (v_1, i_1),(v_2, i_2),….
The reduce function takes the list of pairs, eliminates pairs whose v_i values do not satisfy c, and produces the selection.

b) The map function takes a tuple t and emits the pair (v, i), where v is the value under c and i is the id of the tuple t. The intermediate result is a list of pairs (v_1, i_1),(v_2, i_2),….
The reduce function takes the list of pairs, eliminates pairs whose v_i values do not satisfy c, and produces the projection.

c) The map function takes the product of R and S. It emits the pair (v, i), where v is the value under c and i is the id of the tuple t. The intermediate result is a list of pairs (v_1, i_1),(v_2, i_2),….
The reduce function takes the list of pairs, examines v_i, and emits the joined tuples when v_i values satisfy c.

d) The map function takes R and S. It emits the pair (t, 1), which t is the tuple of t, for each R and S. The intermediate results are two lists of $(t_{r1}, 1)$,$(t_{r2}, 1)$,… and $(t_{s1}, 1)$,$(t_{s2}, 1)$,… from R and S respectively.
The reduce function takes those lists, counts duplicate tuples in each list. It examines the intermediate results from two lists and produces tuples in the larger number of counts when it sees duplicates from both R and S and produces single tuples when it doesn't see duplicate from both relation.

e) The map function takes R and S. It emits the pair (t, 1), which t is the tuple of t, for each R and S. The intermediate results are two lists of $(t_{r1}, 1),(t_{r2}, 1),\dots$ and $(t_{s1}, 1),(t_{s2}, 1),\dots$ from R and S respectively.
The reduce function takes those lists, counts duplicate tuples in each list. It examines the intermediate results from two lists and produces tuples in the smaller number of counts when it only sees duplicates from both R and S.

20.3.1

Define U to be the sum over all sites i of $10du_i$. That is, U is the cost of sending one second's worth of updates to one site other than the site at which the update originated.
Now, consider the value of creating a copy of R at site i. On the positive side, the queries generated at site i that used to have to be answered remotely, at a cost of $10cq_i$, can now be answered for cq_i, a positive benefit of $9cq_i$.
However, the additional copy of R has to be updated. The cost is du_i for the updates generated at site i and $U - 10du_i$ for the updates generated at all other sites. Recall that U represents the cost of a remote update for all generated updates, so if we subtract $10du_i$ we get just the cost of the updates that do not originate at i.
In order for the benefit of creating a copy at i to outweigh the additional update cost, we must have
$9cq_i + 9du_i >= U$
Thus, the optimal selection of sites at which to maintain copies of R is all sites for which the above inequality holds. Roughly, we favor those sites at which there is a lot of activity, but the larger U is (i.e., the greater the update rate), the less likely we are to want to create many copies.
There is one special case: what if the inequality fails for all i; that is, it is not cost effective to create *any* copies. As long as all the q_i's are zero, that's OK. The proverbial ``write-only database'' needs no copies! However, in realistic cases with a heavy update rate, we shall be forced to create one copy. The correct choice is whichever i has the largest value of $cq_i + du_i$.

20.4.1

i) sR

ii) sS

iii) pS + (sR – dR)

iv) pR + (sS – dS)

i) is the best when sR < sS and sR < pS + (sR – dR) to perform the join at site S.
ii) is the best when sR > sS and sS < pR + (sS – dS) to perform the joing at site R.
iii) is the best when sR < sS and sR > pS + (sR – dR)
iv) is the best when sR > sS and sS > pR + (sS – dS)

20.4.2

a)

Let R={A,B}, S={B,C,D}, T={B,E,F}, U={F,G,H}, V={G,I}, and W={H,J}.

The hypergraph is acyclic since there is a full reducer to a single hyperedge U by the ear reductions of R, S, T, V, and W.

b)

After eliminating ears of {A,B}, {B,C,D}, and {G,I}, there is no more ears. Thus there is no way of reducing the hypergraph into a single hyperedge.

c)

Let R={A,B,C,D}, S={A,B,E}, T={B,D,F}, U={C,D,G}, and V={A,C,H}

There exists a full reducer to a single hyperedge R by the ear reductions of S, T, U, and V in any order. Therefore, it’s acyclic.

20.4.3

a) in 20.4.2

S := S $\ltimes$ R

T := T $\ltimes$ S

U := U $\ltimes$ T

U := U $\ltimes$ V

U := U $\ltimes$ W

W := W $\ltimes$ U

V := V $\ltimes$ U

T := T $\ltimes$ U

S := S $\ltimes$ T

R := R $\ltimes$ S

c) in 20.4.2

R := R $\ltimes$ S

R := R $\ltimes$ T

R := R $\ltimes$ U

R := R $\ltimes$ V

S := S $\ltimes$ R

T := T $\ltimes$ R

U := U $\ltimes$ R

V := V $\ltimes$ R

20.4.4

The total number of full reducers of Fig. 20.9 is
3 x 2 x 2 = 12

The first candidates of ear reductions are R, S, and T since U is not an ear yet. Next candidates are the remaining two ears after the first ear reduction. After the second ear reduction U is an ear. Either of two remaining hyperedges can be reduced into a single hyperedge.
Besides the full reducer introduced in Example 20.11, there are 11 more possible full reducers.

20.4.5

True. Eliminating a hyperedge doesn't introduce a new cycle in a hypergraph.

20.4.6

b)intersection and c)difference operations can be improved by the semijoin. One relation in a different site only needs the duplicate portion to produce intersection or difference of two relations. By shipping the semijoin to a different site rather than shipping the entire relation to produce the duplicate portion, the shipping cost can be reduced and algorithm can be improved.
However, for union operation, the entire relation must be shipped to other site anyway. Thus, shipping cost doesn't not have any benefit by the semijoin and there is no algorithm improvement by the semijoin.

20.5.1

a) There is a component *A* at the home computer, and components B and C at each of the banks. Component *B* receives the directive from *A*. *B* checks that there is adequate money in the account, and aborts if not. Otherwise, *B* subtracts $10,000 from the account at *B* and signals *C* to deposit $10,000 into the designated account at *C*. Component *C* checks that the account exists, and aborts if not. Otherwise, *C* adds $10,000 to the account and informs *A* of a successful conclusion.

b) It's possible that it turns out B doesn't have more than $10,000 after it signals C to deposit $10,000. B aborts T but C commits it since there is no problem in C.

c) It's possible that B updates its account subtracting $10,000 and signals C to deposit $10,000 but C is down or disconnected from the network. C doesn't get the signal, thus no transaction occurs at C.
It also can happen that B determines to abort T after it signals C to deposit but C is down or disconnected from the network. C never receives abort T thus commits T.
If either B or C determines to abort T but both B and C are down or disconnected from the network, both never sends or receives the signal thus, one site commits T and the other aborts T.

d) Component A can act as a coordinator described in 20.5.2. Both B and C log the latest decision about the transaction T and sends signals to A. After one or both B and C resume and reconnected to the network, both check the log at A. If the log at A is inconsistent with the local log at B or C, recovers the transaction T by undoing or re-signaling abort T to A. If the log at A is Ready T, then both B and C send the final decisions to A and A drives the recovery.

20.5.2

a) (0,1,P), (0,2,P), (1,0,R), (2,0,D), (0,1,A), (0,2,A)

b) In the first phase, the coordinator exchanges the messages *(0, 1, P)*, *(1, 0, R)* with site 1 and the messages *(0, 2, P)*, *(2, 0, R)* with site 2. These messages may occur in any of 2 * (3 choose 2) = 6 (since (0, 1, R) and (0, 2, R) cannot come before (0, 1, P) and (0, 2, P) respectively) possible inter-leavings. That is, there are four positions in which these messages occur, and we can pick any two of them to devote to the messages involving site 1.
In the second phase, the coordinator can send commit messages to the other two sites in either order. There are thus 6 * 2 = 12 possible message sequences.

c) The number of possible sequences is the same as b). There are (2, 0, D) instead of (2, 0, R) and (0, 1, A) and (0, 2, A) instead of (0, 1, C) and (0, 2, C) in this case.

d) In the phase 1, the coordinator sends the messages (0, 1, P) and (0, 2, P) to site 1 and site 2 respectively. Site 1 replies (1, 0, R) but site 2 doesn't respond due to a failure. Thus, the possible sequences in phase 1 are 2. The coordinator sends (0, 1, A) and (0, 2, A) to both sites with possible sequences are 2. Therefore, the total number of possible sequences is 2 * 2 = 4.

20.5.3

f(n) = (2n choose n) * n

20.6.1

a) Suppose that s, x, and i are the numbers of local shared, exclusive, and increment locks that a transaction needs to have a global lock of that type.
Since no two transactions can hold an exclusive lock on the same element, we need:
$2x > n$

just as for the shared + increment system discussed in the section. We also cannot allow a shared and exclusive lock at the same time, which tells us:
$x + s > n$

Likewise, we cannot have an exclusive and increment lock at the same time, so:
$x + i > n$

Finally, we cannot have a shared and increment lock at the same time, so:

$s + i > n$
For example, we could require that any type of lock requires a majority.

b) Suppose that *s*, *x*, and *u* are the numbers of local shared, exclusive, and update locks that a transaction needs to have a global lock of that type.

$2x > n$

$x + s > n$

,for the same reason in a)

Since no two transactions can hold an update locks on the same element, we need:
$2u > n$

We cannot have an exclusive lock and update lock at the same time, so:
$x + u > n$

c) Suppose that *s*, *x*, *is* and *ix* are the numbers of local shared, exclusive, intention to shared lock and intention to exclusive locks that a transaction needs to have a global lock of that type.

$2x > n$

$x + s > n$

,for the same reason in a) and b)

We cannot have a shared lock and an intention to an exclusive lock at the same time, so:

$ix + s > n$

We cannot have an exclusive lock and an intention to an exclusive lock or an shared lock at the same time, so:

$x + ix > n$

$x + is > n$

20.6.2

a) The 90% of the accesses that are read-only require no messages, since there is a lock table and a copy at each site. The remaining 10% of the accesses require exclusive locks. Thus, each requires three messages between the originating site and each of the four other sites, a total of 12 messages. The average number of messages is 1.2.

b) In majority locking, both read-only and exclusive access require 3 messages for each lock on 2 sites since s = x = |(n+1)/2| and there is no need of messages for its originating site. Thus, the average number of messages is 3 * 2 = 6.

b) In primary-copy locking, only the primary-copy site require no messages for locking and other sites needs to get locks for both shared and exclusive locks from the primary-copy site. Thus, all other 4 sites need to exchange 3 messages for each lock with the primary-copy site. The average number of messages is (3 * 4 * 0.48) = 5.76.

20.7.1

a) N32

b) N1

20.7.2

a) N1

Distance	1	2	4	8	16	32
Node	N8	N8	N8	N14	N21	N38

1 + 1 = 2
1 + 2 = 3
1 + 4 = 5
1 + 8 = 9
1 + 16 = 17
1 + 32 = 33

b) N48

Distance	1	2	4	8	16	32
Node	N51	N51	N56	N56	N1	N21

```
48 + 1 = 49
48 + 2 = 50
48 + 4 = 52
48 + 8 = 56
48 + 16 = 64 => 0           (6-bit mask: 0 ~ 63)
48 + 32 = 80 => 16
```

c) N56

Distance	1	2	4	8	16	32
Node	N1	N1	N1	N1	N8	N32

56 + 1 = 57
56 + 2 = 58
56 + 4 = 60
56 + 8 = 64 => 0
56 + 16 = 72 => 8
56 + 32 = 88 => 24

20.7.3

a) N1 searches for a key that hashes to 27.

N1 -> N38 -> N1

N1 examines its finger table and finds the all entries are below 27 except for N38. So N1 sends a message to N38. N38 finds a key that hashes to 27, so send a message back to the requester N1

b) N1 searches for a key that hashes to 0.

N1 finds 0 is its range, 57,58,…,63,0,1. Thus no messages are sent to other nodes.

c) N51 searches for a key that hashes to 45.

N51 -> N14 -> N48 -> N51

N51 finds that 45 is not in either the successor's range or predecessor's range. N51 checks its finger table and finds all entries are below 45 (logically). So N51 sends a message to N14. N14 finds it is not in its successor's range, so it checks its finger table and finds 45 is in the range of N48. N14 sends a message to N48. N48 sends a message back to N51.

<Finger table of N51>

Distance	1	2	4	8	16	32
Node	N56	N56	N56	N1	N8	N14

51 + 1 = 52
51 + 2 = 53
51 + 4 = 55
51 + 8 = 59
51 + 16 = 67 => 3
51 + 32 = 73 => 9

<Finger table of N14>

Distance	1	2	4	8	16	32
Node	N21	N21	N21	N32	N32	N48

14 + 1 = 15
14 + 2 = 16
14 + 4 = 18
14 + 8 = 22
14 + 16 = 30
14 + 32 = 46

20.7.4

a) 41

Change predecessor and successor links:
1. N41 sets its successor to N42 and its predecessor to nil. The predecessor of N42 remains N38 for the moment.

Stabilize:
2. The predecessor of N_{42} is changed to N_{41}.
3. All key-value pairs whose keys hash to 39 through 41 are moved from N_{42} to N_{41}.
4. When N_{38} eventually runs the stabilization operation, N_{38} changes its successor to N_{41}.
5. N_{38} becomes the predecessor of N_{41}.

b) 62

Change predecessor and successor links:
1. N_{62} sets its successor to N_1 and its predecessor to nil. The predecessor of N_1 remains N_{56} for the moment.

Stabilize:
2. The predecessor of N_1 is changed to N_{62}.
3. All key-value pairs whose keys hash to 57 through 62 are moved from N_1 to N_{62}.
4. When N_{56} eventually runs the stabilization operation, N_{56} changes its successor to N_{62}.
5. N_{56} becomes the predecessor of N_{62}.

20.7.5

Let N_i, N_j, and N_p be a new node to be inserted between N_i and N_p. N_j and N_p to be the successor and the predecessor of N_i respectively.

1. Change predecessor and successor links including predecessor's predecessor and successor's successor.
: A new node N_i sets its successor to N_j and also records the successor of N_j as the successor of its successor.

2. Stabilize:
a. The predecessor of N_j is changed to N_i.
b. The predecessor's predecessor of N_j's successor is changed to N_i.
c. All key-value pairs whose keys hash to p+1 through i are recorded as the key-value pairs of its predecessor in N_j.
d. The key-value pairs in c are moved from N_j to N_i.
e. The key-value pairs in the range of N_j replicated to N_i as its successor's key-value pairs.
f. When N_p eventually runs the stabilization operation, N_p changes its successor to N_i.
g. The successor's successor of N_p's predecessor is changed to N_i.
h. The key-value pairs in the range of N_i replicated to N_p as its successor's key-value pairs.
i. The key-value pairs in the range of N_p replicated to N_i as its predecessor's key-value pairs.