1.Index:

a.

```sql


SELECT DISTINCT p.prod_id
FROM products p
JOIN inventory i
ON p.prod_id = i.prod_id
WHERE i.quan_in_stock < 100;

SELECT DISTINCT p.prod_id
FROM products p
WHERE p.prod_id IN (SELECT prod_id
                        FROM inventory
                        WHERE quan_in_stock < 100);

SELECT * FROM inventory WHERE quan_in_stock > 18;
```

Data Output Messages Notifications

	prod_id [PK] integer	quan_in_stock integer	sales integer
1	1	138	9
2	2	118	19
3	3	228	11
4	4	279	12
5	5	382	13
6	6	109	14
7	7	90	10
8	8	357	6
9	9	215	27
10	10	105	16

```
SELECT * FROM inventory WHERE quan_in_stock > 18;
```

b.

```
14
15  EXPLAIN
16  SELECT * FROM inventory WHERE quan_in_stock > 18;
17  CREATE INDEX idx_hash ON inventory USING HASH(quan_in_stoc
```

Data Output Messages Notifications

	QUERY PLAN text
1	Seq Scan on inventory (cost=0.00..180.00 rows=9595 width=12)
2	Filter: (quan_in_stock > 18)

Total rows: 2 of 2 Query complete 00:00:00.044

```
CREATE INDEX idx_hash ON inventory USING
HASH(quan_in_stock);
EXPLAIN
```

```
SELECT * FROM inventory WHERE quan_in_stock > 18;
```

c.

```
18
19  DROP INDEX IF EXISTS idx_hash;
20
21  CREATE INDEX idx_btree ON inventory USING BTREE(quan_in_st
22  EXPLAIN
23  SELECT * FROM inventory WHERE quan_in_stock > 18;
```

Data Output Messages Notifications

	QUERY PLAN text
1	Seq Scan on inventory (cost=0.00..180.00 rows=9595 width=12)
2	Filter: (quan_in_stock > 18)

Total rows: 2 of 2 Query complete 00:00:00.077

```
CREATE INDEX idx_btree ON inventory USING
BTREE(quan_in_stock);
EXPLAIN
```

```
SELECT * FROM inventory WHERE quan_in_stock > 18;
```

d. There is no differenece between query plans in b and c. Because numbers of products having quan_in_stock > 18 are very large so the DBMS considered seq scan as fastest way.

e.

```
DROP INDEX IF EXISTS idx_hash;
DROP INDEX IF EXISTS idx_btree;
```

Data Output | Messages | Notifications

```
DROP INDEX

Query returned successfully in 105 msec.
```

Total rows: 2000 of 10000 | Query complete 00:00:00.105

```
DROP INDEX IF EXISTS idx_hash;
DROP INDEX IF EXISTS idx_btree;
```

redo b

```
14
15  EXPLAIN
16  SELECT * FROM inventory WHERE quan_in_stock > 400;
17  CREATE INDEX idx_hash ON inventory USING HASH(quan_in_stoc
18
19  CREATE INDEX idx_btree ON inventory USING BTREE(quan_in_st
20  EXPLAIN
21  SELECT * FROM inventory WHERE quan_in_stock > 18;
22
23  SELECT * FROM inventory
24
25  DROP INDEX IF EXISTS idx_hash;
26  DROP INDEX IF EXISTS idx_btree;
```

Data Output Messages Notifications

	QUERY PLAN text
1	Seq Scan on inventory (cost=0.00..180.00 rows=2020 width=12)
2	Filter: (quan_in_stock > 400)

Total rows: 2 of 2 Query complete 00:00:00.099

redo c

```sql
EXPLAIN
SELECT * FROM inventory WHERE quan_in_stock > 400;
CREATE INDEX idx_hash ON inventory USING HASH(quan_in_stoc

CREATE INDEX idx_btree ON inventory USING BTREE(quan_in_st
EXPLAIN
SELECT * FROM inventory WHERE quan_in_stock > 18;

SELECT * FROM inventory

DROP INDEX IF EXISTS idx_hash;
DROP INDEX IF EXISTS idx_btree;
```

Data Output Messages Notifications

	QUERY PLAN text
1	Bitmap Heap Scan on inventory (cost=27.94..108.19 rows=2020 width=12)
2	Recheck Cond: (quan_in_stock > 400)
3	-> Bitmap Index Scan on idx_btree (cost=0.00..27.43 rows=2020 width=0)
4	Index Cond: (quan_in_stock > 400)

Total rows: 4 of 4 Query complete 00:00:00.041

redo d

Now, there are differences between b and c when redo because numbers of products having quan_in_stock > 400 are fewer.
Hash indexes only support equality comparisons when performing lookups. Bitmap index scan is useful with low cardinality and full match value.

f. There are differences between d and e when list of records in Inventory having "quan_in_stock" greater than 400 and 18. Because numbers of products having quan_in_stock > 400 are fewer than numbers of products having quan_in_stock > 18.

2. Query:
a.

```sql
SELECT DISTINCT p.prod_id
FROM products p
JOIN inventory i
ON p.prod_id = i.prod_id
WHERE i.quan_in_stock < 100;

SELECT DISTINCT p.prod_id
FROM products p
WHERE p.prod_id IN (SELECT prod_id
                    FROM inventory
                    WHERE quan_in_stock < 100);
```

Data Output Messages Notifications

	prod_id [PK] integer
1	1269
2	652
3	6430
4	951
5	1898
6	70
7	5843
8	8174
9	8034
10	7662

```sql
SELECT DISTINCT p.prod_id
FROM products p
JOIN inventory i
ON p.prod_id = i.prod_id
WHERE i.quan_in_stock < 100;
```

```sql
SELECT DISTINCT p.prod_id
FROM products p
JOIN inventory i
ON p.prod_id = i.prod_id
WHERE i.quan_in_stock < 100;

SELECT DISTINCT p.prod_id
FROM products p
WHERE p.prod_id IN (SELECT prod_id
                    FROM inventory
                    WHERE quan_in_stock < 100);
```

Data Output Messages Notifications

	prod_id [PK] integer
1	1269
2	652
3	6430
4	951
5	1898
6	70
7	5843
8	8174
9	8034
10	7662

```sql
SELECT DISTINCT p.prod_id
FROM products p
WHERE p.prod_id IN (SELECT prod_id
                    FROM inventory
                    WHERE quan_in_stock < 100);
```

b.

```sql
EXPLAIN
SELECT DISTINCT p.prod_id
FROM products p
JOIN inventory i
ON p.prod_id = i.prod_id
WHERE i.quan_in_stock < 100;

EXPLAIN
SELECT DISTINCT p.prod_id
FROM products p
WHERE p.prod_id IN (SELECT prod_id
```

Data Output Messages Notifications

	QUERY PLAN text
1	HashAggregate (cost=437.67..457.94 rows=2027 width=4)
2	Group Key: p.prod_id
3	-> Hash Join (cost=205.34..432.60 rows=2027 width=4)
4	Hash Cond: (p.prod_id = i.prod_id)
5	-> Seq Scan on products p (cost=0.00..201.00 rows=10000 width=4)
6	-> Hash (cost=180.00..180.00 rows=2027 width=4)
7	-> Seq Scan on inventory i (cost=0.00..180.00 rows=2027 width=4)
8	Filter: (quan_in_stock < 100)

Total rows: 8 of 8 Query complete 00:00:00.096

```sql
ON p.prod_id = i.prod_id
WHERE i.quan_in_stock < 100;

EXPLAIN
SELECT DISTINCT p.prod_id
FROM products p
WHERE p.prod_id IN (SELECT prod_id
                    FROM inventory
                    WHERE quan_in_stock < 100);

EXPLAIN
SELECT * FROM inventory WHERE quan_in_stock > 400;
```

Data Output Messages Notifications

	QUERY PLAN text
1	HashAggregate (cost=437.67..457.94 rows=2027 width=4)
2	Group Key: p.prod_id
3	-> Hash Join (cost=205.34..432.60 rows=2027 width=4)
4	Hash Cond: (p.prod_id = inventory.prod_id)
5	-> Seq Scan on products p (cost=0.00..201.00 rows=10000 width=4)
6	-> Hash (cost=180.00..180.00 rows=2027 width=4)
7	-> Seq Scan on inventory (cost=0.00..180.00 rows=2027 width=4)
8	Filter: (quan_in_stock < 100)

Total rows: 8 of 8 Query complete 00:00:00.054

No difference between 2 ways of query. Because there is no index in quan_in_stock (inventory), and we have index on prod_id of table products

c.

after create index btree on quan_in_stock in inventory table, it is used in both ways of query.

```sql
EXPLAIN
SELECT DISTINCT p.prod_id
FROM products p
JOIN inventory i
ON p.prod_id = i.prod_id
WHERE i.quan_in_stock < 100;

EXPLAIN
SELECT DISTINCT p.prod_id
FROM products p
WHERE p.prod_id IN (SELECT prod_id
                    FROM inventory
```

Data Output Messages Notifications

	QUERY PLAN text
1	HashAggregate (cost=366.00..386.27 rows=2027 width=4)
2	Group Key: p.prod_id
3	-> Hash Join (cost=133.67..360.93 rows=2027 width=4)
4	Hash Cond: (p.prod_id = i.prod_id)
5	-> Seq Scan on products p (cost=0.00..201.00 rows=10000 width=4)
6	-> Hash (cost=108.33..108.33 rows=2027 width=4)
7	-> Bitmap Heap Scan on inventory i (cost=27.99..108.33 rows=2027 width=4)
8	Recheck Cond: (quan_in_stock < 100)
9	-> Bitmap Index Scan on idx_btree (cost=0.00..27.49 rows=2027 width=0)
10	Index Cond: (quan_in_stock < 100)

Total rows: 10 of 10 Query complete 00:00:00.056

```sql
ON p.prod_id = i.prod_id
WHERE i.quan_in_stock < 100;

EXPLAIN
SELECT DISTINCT p.prod_id
FROM products p
WHERE p.prod_id IN (SELECT prod_id
                    FROM inventory
                    WHERE quan_in_stock < 100);

EXPLAIN
SELECT * FROM inventory WHERE quan_in_stock > 400;
CREATE INDEX idx_hash ON inventory USING HASH(quan_in_stoc
```

Data Output Messages Notifications

	QUERY PLAN text
1	HashAggregate (cost=366.00..386.27 rows=2027 width=4)
2	Group Key: p.prod_id
3	-> Hash Join (cost=133.67..360.93 rows=2027 width=4)
4	Hash Cond: (p.prod_id = inventory.prod_id)
5	-> Seq Scan on products p (cost=0.00..201.00 rows=10000 width=4)
6	-> Hash (cost=108.33..108.33 rows=2027 width=4)
7	-> Bitmap Heap Scan on inventory (cost=27.99..108.33 rows=2027 width=4)
8	Recheck Cond: (quan_in_stock < 100)
9	-> Bitmap Index Scan on idx_btree (cost=0.00..27.49 rows=2027 width=0)
10	Index Cond: (quan_in_stock < 100)

Total rows: 10 of 10 Query complete 00:00:00.067

Because numbers of products having quan_in_stock < 100 are not much. Bitmap index scan is useful with low cardinality and full match value.

3. Trigger:

```
27
28  CREATE OR REPLACE FUNCTION update_inventory()
29  RETURNS TRIGGER AS $$
30  BEGIN
31    IF TG_OP = 'INSERT' THEN
32      UPDATE inventory SET quan_in_stock = quan_in_stock - N
33      WHERE prod_id = NEW.prod_id;
34    ELSIF TG_OP = 'UPDATE' THEN
35      UPDATE inventory SET quan_in_stock = quan_in_stock + O
36      WHERE prod_id = NEW.prod_id;
37    END IF;
38    RETURN NEW;
39  END;
40  $$ LANGUAGE plpgsql;
41
42  CREATE TRIGGER update_trigger
43  AFTER INSERT OR UPDATE ON orderlines
44  FOR EACH ROW
45  EXECUTE FUNCTION update_inventory();
```

Data Output | Messages | Notifications

```
CREATE TRIGGER

Query returned successfully in 38 msec.
```

Total rows: 4 of 4 | Query complete 00:00:00.038

```
CREATE OR REPLACE FUNCTION update_inventory()
RETURNS TRIGGER AS $$
BEGIN
  IF TG_OP = 'INSERT' THEN
    UPDATE inventory SET quan_in_stock =
quan_in_stock - NEW.quantity, sales = sales +
NEW.quantity
    WHERE prod_id = NEW.prod_id;
  ELSIF TG_OP = 'UPDATE' THEN
    UPDATE inventory SET quan_in_stock =
quan_in_stock + OLD.quantity - NEW.quantity, sales
= sales + NEW.quantity - OLD.quantity
    WHERE prod_id = NEW.prod_id;
  END IF;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER update_trigger
AFTER INSERT OR UPDATE ON orderlines
FOR EACH ROW
EXECUTE FUNCTION update_inventory();
```

4. Function:

```sql
CREATE OR REPLACE FUNCTION list_product(limits INT)
RETURNS TABLE(prod_id INT, category INT)
LANGUAGE plpgsql
AS
$$
BEGIN
  RETURN QUERY
  SELECT p.prod_id, p.category
  FROM products p
  JOIN inventory i ON i.prod_id = p.prod_id
  WHERE i.quan_in_stock < limits;
END;
$$
```

Data Output | Messages | Notifications

```
CREATE FUNCTION

Query returned successfully in 48 msec.
```

```sql
DROP FUNCTION IF EXISTS list_product;

CREATE OR REPLACE FUNCTION list_product(limits INT)
RETURNS TABLE(prod_id INT, category INT)
LANGUAGE plpgsql
AS
$$
BEGIN
  RETURN QUERY
  SELECT p.prod_id, p.category
  FROM products p
  JOIN inventory i ON i.prod_id = p.prod_id
  WHERE i.quan_in_stock < limits;
END;
$$

SELECT * FROM list_product(50);
```

Data Output Messages Notifications

	prod_id integer	category integer
1	12	7
2	24	3
3	39	8
4	59	7
5	74	2
6	84	3
7	117	13
8	122	9
9	135	5
10	138	10