A.

Ejecutamos explain sobre la base de datos limpia, obteniendo los siguientes costes:

	QUERY PLAN text
1	Aggregate (cost=4869.12..4869.13 rows=1 width=0)
2	-> HashAggregate (cost=4869.10..4869.11 rows=1 width=4)
3	Group Key: orders.customerid
4	-> HashAggregate (cost=4869.08..4869.09 rows=1 width=4)
5	Group Key: orders.customerid
6	-> Seq Scan on orders (cost=0.00..4868.33 rows=303 width=4)
7	Filter: ((totalamount >= '100'::numeric) AND (orderdate >= '2014-04-01'::date) AND (orderdate <= '2014-04-30'::date))

Tras analizar el resultado consideramos que un buen cantdidato a índice sería la columna orderdate. Este es el resultado tras crear el índice:

Data Output | Explain | Messages | History

	QUERY PLAN text
1	Aggregate (cost=1503.98..1503.99 rows=1 width=0)
2	-> HashAggregate (cost=1503.96..1503.97 rows=1 width=4)
3	Group Key: orders.customerid
4	-> HashAggregate (cost=1503.95..1503.96 rows=1 width=4)
5	Group Key: orders.customerid
6	-> Bitmap Heap Scan on orders (cost=21.59..1503.19 rows=303 width=4)

Vemos que el coste de la consulta disminuye hasta casi cuatro veces.
Para elegir este índice hicimos varias pruebas, con orderdate, customerid y la combinación de ambos campos. El que mejor resultado nos dio fue orderdate.

B.

Todas las consultas a la base de datos se han realizado con el índice creado. Con la consulta preparada hemos obtenido un resultado ligeramente mejor, pero no demasiado, puesto que se tiene que realizar el bindParam cada vez que aumentamos el umbral.

C.

La primera consulta debe recorrer por completo dos tablas: “customers” y “orders”. Primero recorre “orders” buscando aquellas entradas con status=’Paid’ y despúes recorre “customers” seleccionando los id que no estuviesen en la tabla anterior.

La segunda consulta debe recorrer también ambas tablas, pero esta vez solo se impone un filtro a la tabla “orders”. Posteriormente tras hacer la unión de ambas consultas se recorre el resultado una vez más aplicando el “group by”. Por este movito el coste de esta consulta es mayor.

Por último la tercera consulta también recorre ambas tablas por completo, haciendo la resta de conjuntos de ambas. Esta consulta nos parece muy similar a la primera, no obstante el coste de ella es el mayor de las tres.

D.

Ejecutamos explain sobre la primera consulta y obtenemos la siguiente salida:

Data Output | Explain | Messages | History

	QUERY PLAN text
1	Aggregate (cost=3507.17..3507.18 rows=1 width=8)
2	-> Seq Scan on orders (cost=0.00..3504.90 rows=909 width=0)
3	Filter: (status IS NULL)

Como podemos observar la tabla “orders” se recorre por completo recogiendo solo aquellas filas en las que el status tiene un valor nulo.

El resultado de explain sobre la segunda consulta es:

Data Output | Explain | Messages | History

	QUERY PLAN text
1	Aggregate (cost=3961.65..3961.66 rows=1 width=8)
2	-> Seq Scan on orders (cost=0.00..3959.38 rows=909 width=0)
3	Filter: ((status)::text = 'Shipped'::text)

De nuevo se recorre toda la tabla “orders”. Sin embargo en este caso, el coste es bastante mayor porque ahora debe ir comparando cadenas de caracteres para aplicar el filtro, mientras que en la otra solo comprueba que no haya valor en el campo.

Creamos el index en la columna status y volvemos a ejecutar los dos explain:

	QUERY PLAN text
1	Aggregate (cost=1496.52..1496.53 rows=1 width=8)
2	-> Bitmap Heap Scan on orders (cost=19.46..1494.25 rows=909 width=0)
3	Recheck Cond: (status IS NULL)
4	-> Bitmap Index Scan on i_status (cost=0.00..19.24 rows=909 width=0)
5	Index Cond: (status IS NULL)

Data Output | Explain | Messages | History

	QUERY PLAN text
1	Aggregate (cost=1498.79..1498.80 rows=1 width=8)
2	-> Bitmap Heap Scan on orders (cost=19.46..1496.52 rows=909 width=0)
3	Recheck Cond: ((status)::text = 'Shipped'::text)
4	-> Bitmap Index Scan on i_status (cost=0.00..19.24 rows=909 width=0)
5	Index Cond: ((status)::text = 'Shipped'::text)

Podemos ver que el coste de ambas consultas tras crear el índice es prácticamente el mismo, además de haberse reducido a más de la mitad del coste sin el índice.

Ejecutamos analyze y volvemos a ver el plan de ejecucución de las consultas:

	QUERY PLAN text
1	Aggregate (cost=7.25..7.26 rows=1 width=8)
2	-> Index Only Scan using i_status on orders (cost=0.42..7.24 rows=1 width=0)
3	Index Cond: (status IS NULL)

	QUERY PLAN text
1	Aggregate (cost=4278.69..4278.70 rows=1 width=8)
2	-> Seq Scan on orders (cost=0.00..3959.38 rows=127726 width=0)
3	Filter: ((status)::text = 'Shipped'::text)

EL coste de la primera consulta ha disminuido mucho, mientras que el de la segunda ha aumentado. Parece ser que tras ejecutar analyze, en la primera consulta se sigue emplendo en índice pero en la segunda no, pues recorre la tabla entera.

Comparamos con las dos nuevas consultas:

	QUERY PLAN text
1	Aggregate (cost=2322.24..2322.25 rows=1 width=8)
2	-> Bitmap Heap Scan on orders (cost=361.73..2276.66 rows=18234 width=0)
3	Recheck Cond: ((status)::text = 'Paid'::text)
4	-> Bitmap Index Scan on i_status (cost=0.00..357.18 rows=18234 width=0)
5	Index Cond: ((status)::text = 'Paid'::text)

Vemos que los costes para ambas consultas son muy similares y que en ellas se hace uso del índice

	QUERY PLAN text
1	Aggregate (cost=2930.58..2930.59 rows=1 width=8)
2	-> Bitmap Heap Scan on orders (cost=706.11..2841.00 rows=35831 width=0)
3	Recheck Cond: ((status)::text = 'Processed'::text)
4	-> Bitmap Index Scan on i_status (cost=0.00..697.15 rows=35831 width=0)
5	Index Cond: ((status)::text = 'Processed'::text)

creado anteriormente.

Tras ejecutar las consultas sin explain y ver el resultado nos hemos dado cuenta de que no existe ninguna fila con status a NULL y que la gran mayoría de filas de la tabla tiene status “Shipped”. Por tanto creemos que tras ejecutar analyze postgres decide que es mejor recorrer la tabla entera para las consultas con status=’Shipped’ y usar la búsqueda binaria para el resto de valores de status.